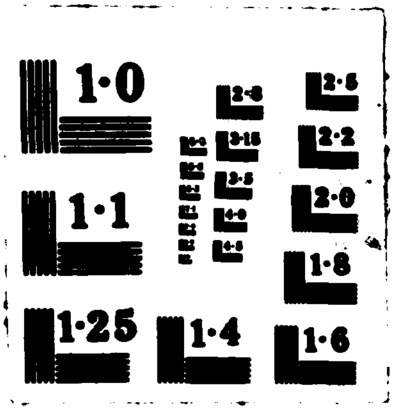
AD-A187 033 NESTED TRANSACTIONS CONFLICT-BASED LOCKING AND DYNAMIC 1/1
ATOMICITY(U) MASSACHUSETTS INST OF TECH CAMBRIDGE LAB
FOR COMPUTER SCIENCE A FEKETE ET AL SEP 87
UNCLASSIFIED MIT/LCS/TM-340 N00014-85-K-0168 F/G 12/7 NL


1·0
2·8
2·5
3·2
3·6
2·2
4·0
2·0
1·1
1·8
1·25
1·4
1·6

LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-340

NESTED TRANSACTIONS, CONFLICT-BASED LOCKING, AND DYNAMIC ATOMICITY

Alan Fekete
Nancy Lynch
Michael Merritt
William Weihl

September 1987

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

ADA187033

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TM-340	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-85-K-0168 & N00014-83-K-0125
6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science 6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22217	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)
NESTED TRANSACTIONS, CONFLICT-BASED LOCKING, AND DYNAMIC ATOMICITY

12 PERSONAL AUTHOR(S)
Fekete, Alan; Lynch, Nancy; Merritt, Michael; and Weihl, William

13a TYPE OF REPORT Technical	13b TIME COVERED FROM ______ TO ______	14 DATE OF REPORT (Year, Month, Day) 1987 September	15 PAGE COUNT 53

16 SUPPLEMENTARY NOTATION

17 COSATI CODES

FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
nested transactions, atomic actions, concurrency control, recovery, databases, serializability, commutative operations, locking rules

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

In this paper ~~we~~ examines some concurrency control algorithms for nested transaction systems. ~~We define~~ a simple local property called dynamic atomicity for data objects in such a system, and we show that all the executions of a system are serially correct (despite concurrency and transaction aborts) provided each data object is dynamic atomic. ~~We then apply~~ this result is applied to give a correctness proof for a new algorithm, called Conflict-Based Locking, for managing data in a nested transaction system. The algorithm uses a table specifying which operations may not proceed concurrently to determine when an operation may be granted a lock on an object. The algorithm is an extension of a general conflict-based locking protocol introduced by Weihl for transaction systems without nesting. It is also similar to Moss' algorithm, which uses read- and write-locks and a stack of versions of each object to ensure the serializability and recoverability of transactions accessing the data. We show that objects implemented using Moss' algorithm of the new Conflict-Based Locking algorithm are dynamic atomic. Thus it follows that if

to next page

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b TELEPHONE (Include Area Code) (617) 253-5894 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR
83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

U.S. Government Printing Office: 1985-507-047

Unclassified

19. each object in a nested transaction system uses either Moss' algorithm or the new Conflict-Based Locking algorithm, then all executions of the system will be serially correct.

Nested Transactions, Conflict-Based Locking, and Dynamic Atomicity

Alan Fekete[1]

Nancy Lynch[2]

Michael Merritt[3]

William Weihl[4]

Abstract: In this paper we examine some concurrency control algorithms for nested transaction systems. We define a simple local property called *dynamic atomicity* for data objects in such a system, and we show that all the executions of a system are serially correct (despite concurrency and transaction aborts) provided each data object is dynamic atomic. We then apply this result to give a correctness proof for a new algorithm, called *Conflict-Based Locking*, for managing data in a nested transaction system. The algorithm uses a table specifying which operations may not proceed concurrently to determine when an operation may be granted a lock on an object. The algorithm is an extension of a general conflict-based locking protocol introduced by Weihl for transaction systems without nesting. It is also similar to Moss' algorithm, which uses read- and write-locks and a stack of versions of each object to ensure the serializability and recoverability of transactions accessing the data. We show that objects implemented using Moss' algorithm or the new Conflict-Based Locking algorithm are dynamic atomic. Thus it follows that if each object in a nested transaction system uses either Moss' algorithm or the new Conflict-Based Locking algorithm, then all executions of the system will be serially correct.

Keywords: nested transactions, atomic actions, concurrency control, recovery, databases, serializability, commutative operations, locking rules.

September 8, 1987

[1] Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass.

[2] Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass.

[3] AT&T Bell Laboratories, Murray Hill, New Jersey.

[4] Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass.

Table of Contents

Nested Transactions, Conflict-Based Locking, and Dynamic Atomicity[5]

1. Introduction

A major part of database research over several years has been the design and analysis of algorithms to maintain consistent data in the face of interleaved accesses, aborts of operations, replication of information and failures of system components. The most popular and simple protocol is two phase locking with separate read and write locks; other methods include arbitrary conflict-based locking, timestamp-based techniques, and locking that uses special structure of the data (e.g. a hierarchical arrangement) ([T],[We] and many others). A powerful theory has been developed to prove the correctness of these algorithms, based on the idea that a protocol is correct if it ensures that all executions are equivalent to serial executions [EGLT],[P],[BG]. This theory proves serializability by showing that a precedence graph contains no cycles.

Recently, some ideas in database system design and more general distributed system design have led several research groups to study the possibility of giving more structure to the transactions that are the basic unit of atomicity. When a transaction can contain concurrent operations that are to be performed atomically, or operations that can be aborted independently, we say that the operations form *subtransactions* of the original transaction. Thus we consider a system where transactions can be *nested*. This idea was first suggested by Davies under the name *spheres of control* [D]. A primitive example of this concept is implemented in System R, where a transaction can be restarted at the last savepoint. In general distributed systems like Argus [LiS,LHJLSW] or Clouds [A], the basic services are often provided by Remote Procedure Calls which, at their best ("At Most Once" semantics), are atomic. Since providing a service will often require using other services, the transactions that implement services ought to be nested.

The implementation of a nested transaction system requires extending the algorithms that have previously been considered for concurrency control, recovery and replication. The work of Reed [R]

[5] The work of the second author (and through her, the work of the first author) was supported in part by the Office of Naval Research under Contract N00014-85-K-0168, by the Office of Army Research under contract DAAG29-84-K-0058, by the National Science Foundation under Grants MCS-8306854, DCR-83-02391, and CCR-8611442, and by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125. The work of the fourth author was supported in part by the National Science Foundation under Grant DCR-8510014, by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125, and by an IBM Faculty Development Award.

extended multi-version timestamp concurrency control to provide nested transaction data management. Moss [Mo] extended two phase locking with separate read and write locks to handle nesting, and this algorithm is the basis of data management in the Argus system implemented at MIT.

In this paper we develop a powerful method for showing the correctness of concurrency control algorithms for nested transaction systems. We introduce a simple local property, which we call *dynamic atomicity,* and we show that a system is serially correct for non-orphan transactions (as defined for nested systems in [LM]) provided each object in the system is dynamic atomic. The corresponding definitions and results for systems without nesting were first given by Weihl [We]. Our discussion covers both concurrency control and recovery from aborts. However, we do not consider all the failure cases that the real system must deal with, as our model does not yet include crashes that compromise the system state.

To illustrate the power of these ideas, we introduce here a new algorithm for concurrency control in nested transaction systems. Our algorithm allows more concurrency than Moss' algorithm by using semantic information about conflicts between operations of abstract data objects. As an example of such semantic information, consider a data object that represents a bank account, with operations DEPOSIT(AMT), WITHDRAW(AMT), and BALANCE. The DEPOSIT operations modify the state of the object, so ordinary Read/Write locking would not allow two DEPOSIT operations to run concurrently. However, there is really no harm in allowing this concurrency if the operations are implemented appropriately (for example, by recording an "intention to deposit" in the object's history) since the final balance is independent of the order in which the deposits are performed. This property is usually described by saying that the two DEPOSIT actions "commute", and that therefore their locks need not "conflict". The algorithm we give combines the techniques of Moss' algorithm with the arbitrary conflict-based (but un-nested) locking algorithm of Weihl [We].

We prove that any object implemented using our "Conflict-Based Locking" algorithm is dynamic atomic, and thus that a system in which the objects are implemented this way is serially correct. We also show that objects implemented using Moss' algorithm for Read/Write Locking are dynamic atomic. We believe that other concurrency control techniques can also be proved to yield dynamic atomic objects. The results in this paper show that a nested transaction system is serially correct if each object is implemented using a concurrency control mechanism that guarantees dynamic atomicity.

This paper is part of a major research effort to offer clean, readable descriptions of algorithms for managing data in a nested trasaction system, together with rigorous proofs of the correctness of these algorithms. Other parts of the project include studying replicated data management algorithms [GL], orphan elimination algorithms [HLMW], and concurrency control using timestamps [As]. All this work is based on a simple model of concurrent systems using *I/O automata* and an operational style of reasoning about their schedules. The first fruits of this program are detailed in [LM], which proves the correctness of exclusive locking, and provides a basic framework for presenting the ideas of this paper.

In this paper, we first review the I/O automaton model of computation. This is similar to models like Communicating Sequential Processes [Ho], in that automata interact by synchronizing on shared operations. The main difference from other models is that we distinguish the input and output operations of each automaton. Any operation shared between components of a system can be an output of at most one component, and that component is in control of the operation, because no automaton is allowed to refuse to execute an input. Though automata have states as well as operations, we concentrate our analysis on the sequence of operations performed (the *schedule* of the system) - this operational mode of reasoning is quite different from assertional invariant methods used elsewhere in reasoning about distributed systems, but we find it very powerful and yet simple for the set of problems we consider.

Next, we show how to use I/O automata to model the parts of a nested transaction system. Each transaction process is represented by an automaton, as is each data object. The actions of calling a subtransaction, invoking an access to an object, and returning a result are each split into two operations, one requesting the action and one delivering the request to the recipient. The request operation is an output of the caller and an input to the controller (which acts as a communication system) while the delivery operation is an output of the controller and an input of the recipient. Thus, each transaction (and each object) shares operations only with the controller. A *serial system* is the result of composing transaction and object automata with a very restricted controller called the *serial scheduler*, which runs the subtransactions of any transaction sequentially (with no concurrency between siblings) and only aborts transactions before they start running. The serial scheduler is very simple to understand and is used as the basis of our correctness condition.

We then introduce a *generic system* to model a system that allows concurrency and aborts of running transactions, and uses some (unspecified) concurrency control and recovery mechanism for each data object. We use a new sort of I/O automaton called a *generic object*, which is like the object automaton of the serial system, but which receives information about the fate of transactions so that it can continue to respond correctly to invocations of operations. We also use a different controller called a *generic controller*, which transmits requests to the appropriate recipient with arbitrary delay, allowing siblings to run concurrently or to abort after performing some work. A generic system is the result of composing the transaction automata, generic objects and generic controller.

We define a simple property of generic objects, which we call *dynamic atomicity*. Dynamic atomic objects respond to accesses in a way that ensures that the accesses can be serialized in an order that is determined dynamically as the transactions return. We show that if every generic object in a generic system is dynamic atomic, then the system is correct in the sense (first suggested in [LM]) that each transaction that does not have an aborted ancestor is unable to tell whether it is running in the generic system or in a corresponding serial system. The proof proceeds by taking a schedule of such a generic system, choosing a subsequence of operations containing the operations of the selected transaction and all the operations that could have affected them, and rearranging that subsequence in any order compatible with a few simple conditions. The resulting sequence is shown to be a schedule of a serial system that appears the same to the transaction chosen. We also define a property called *local-dynamic atomicity*, which implies dynamic atomicity, and is more easily decidable from the schedules of the object involved.

We next provide a simple semantic definition for the key notion of *commutativity*. This definition in turn relies on a definition of *equieffective schedules* to describe schedules that are "observationally indistinguishable". In order to implement our new locking algorithm, we will assume that for each object we are given a table listing which operations conflict, and we require that any operations that do not conflict be commutative.

We formalize our algorithm for conflict-based locking by constructing a *Conflict-Based Locking object* W(X) for each basic object X. W(X) maintains lock tables and information about previous operations. and delays responding to invocations until the locking rules permit the required locks to be granted. We prove that any Conflict-Based Locking object is local-dynamic atomic. Thus a generic system in which every generic object is a Conflict-Based Locking object is serially correct. We also show that the R/W locking objects of [FLMW], which use Moss' algorithm for concurrency control, behave just like Conflict-Based Locking objects for a suitable choice of conflict table, and are thus local-dynamic atomic. Therefore a generic system where each object is either a R/W locking object or a Conflict-Based Locking object is serially correct. In particular, the correctness of a system such as Argus, where every object is implemented using Moss' algorithm, follows from the results in this paper.

There have been several other attempts to provide rigorous proofs of the correctness of algorithms for data management in nested transaction systems. The first was [Ly], which presented a model that successfully handled exclusive locking, but which proved difficult to extend to more complicated problems such as orphan elimination [Go]. The main deficiencies of this earlier model seem to be the lack of distinction between inputs and outputs, and the lack of explicit representations for transactions and their interfaces. These deficiencies were remedied in [LM], where the operational model discussed above was defined; this paper again proved correctness of exclusive locking. Other work following [LM] includes [HLMW], [GL], [FLMW], and [As]. This paper continues the work of [LM] by presenting and verifying an algorithm that allows arbitrary types of locks. A different program to study concurrency control in nested transaction systems has been offered in [BBGLS,BBG], where a major motivation is to analyze protocols that operate on data at different levels of abstraction, but where recovery is not considered. The argument for the correctness of Moss' algorithm in [BBG] considers only the locking rules and not the state maintenance methods, so correctness is proved only in the absence of aborts. Concurrency control and recovery algorithms are also analyzed in [MGG], which is concerned mainly with levels of abstraction.

This paper uses many concepts from [LM], but we have repeated everything needed to make it self-contained, and indicated where definitions or details differ.

2. I/O Automata

The following is a brief introduction to a model that is described in [LM] and developed at length, with extensions to express infinite behavior, in [LT].

All components in our systems, transactions, objects and controllers, will be modelled by *I/O automata*. An I/O automaton $\mathcal{A}$ has a set of *states*, some of which are designated as *initial states*. It has *operations*, each classified as either an *input operation* or an *output operation*. Finally, it has a transition relation, which is a set of triples of the form (s',π,s), where s' and s are states, and π is an operation. This triple means that in state s', the automaton can atomically do operation π and change to state s. An element of the transition relation is called a *step* of the automaton. The output operations are intended to model the actions that are triggered by the automaton itself, while the input operations model the actions that are triggered by the environment of the automaton.

Given a state s' and an operation π, we say that π is *enabled* in s' if there is a state s for which (s',π,s) is a step. We require the following condition:

Input Condition: Each input operation π is enabled in each state s'.
This condition says that an I/O automaton must be prepared to receive any input operation at any time.

An *execution* of $\mathcal{A}$ is an finite alternating sequence $s_0,\pi_1,s_1,\pi_2,\ldots,\pi_n,s_n$ of states and operations of $\mathcal{A}$, beginning and ending with a state. Furthermore, s_0 is a start state of $\mathcal{A}$, and each triple (s',π,s) that occurs as a consecutive subsequence is a step of $\mathcal{A}$. From any execution, we can extract the *schedule*, which is the subsequence of the execution consisting of operations only. Because transitions to different states may have the same operation, different executions may have the same schedule. We say that a schedule α of $\mathcal{A}$ *can leave* $\mathcal{A}$ *in* state s if there is some execution of $\mathcal{A}$ with schedule α and final state s. We say that an operation π is *enabled after* a schedule α of $\mathcal{A}$ if there exists a state s such that α can leave $\mathcal{A}$ in state s and π is enabled in s. Since the same operation may occur several times in an execution or schedule, we refer to a single occurrence of an operation as an *event*.

We describe systems as consisting of interacting components, each of which is an I/O automaton. It is convenient and natural to view systems as I/O automata, also. Thus, we define a composition operation for I/O automata to yield a new I/O automaton. A set of I/O automata may be composed to create a *system* $\mathcal{S}$ if the sets of output operations of the various automata are pairwise disjoint. (Thus, every output operation in $\mathcal{S}$ will be triggered by exactly one component.) A state of the composed automaton is a tuple of states, one for each component, and the start states are tuples consisting of start states of the components. The operations of the composed automaton are those of the component automata. Thus, each operation of the composed automaton is an operation of a subset of the set of component automata. An operation is an output of the composed automaton exactly if it is an output of some component. (The output operations of a system are intended to be exactly those that are triggered by components of the system, while the input operations of a system are those that are triggered by the system's environment.) During an operation π of a composed automaton, each of the components that has operation π carries out the operation, while the remainder stay in the same state.

An *execution* or *schedule* of a system is defined to be an execution or schedule of the automaton composed of the individual automata of the system. If α is a schedule of a system with component $\mathcal{A}$, then the projection of α on $\mathcal{A}$, denoted by $\alpha|\mathcal{A}$, is the subsequence of α containing all the operations of $\mathcal{A}$. Clearly, $\alpha|\mathcal{A}$ is a schedule of $\mathcal{A}$.

The following lemma from [LM] expresses formally the idea that an operation is under the control of the component of which it is an output.

Lemma 1: Let α' be a schedule of a system $\mathcal{S}$, and let $\alpha = \alpha'\pi$, where π is an output operation of component $\mathcal{A}$. If $\alpha|\mathcal{A}$ is a schedule of $\mathcal{A}$, then α is a schedule of $\mathcal{S}$.

Proof: Since $\alpha|\mathcal{A}$ is a schedule of $\mathcal{A}$, there is an execution β of $\mathcal{A}$ with schedule $\alpha|\mathcal{A}$. Let β' be the execution of $\mathcal{A}$ consisting of all but the last step of β. Similarly, since α' is a schedule of $\mathcal{S}$, there is an execution γ of $\mathcal{S}$ with schedule α'. It is possible that $\mathcal{A}$ has an execution in γ that is different from β', since different executions may have the same schedule. But it is easy to show, by induction on the length of γ, that there is another execution γ' of $\mathcal{S}$ in which component $\mathcal{A}$ has execution β', and which is otherwise identical to γ. The schedule of γ' is α'. Since π is not an output operation of any other component, π is defined from the state reached at the end of γ', so that $\alpha = \alpha'\pi$ is a schedule of $\mathcal{S}$. □

We say that automaton $\mathcal{A}$ *preserves* a property P of schedules of $\mathcal{A}$ if $\alpha = \alpha'\pi$ satisfies P whenever α is a schedule of $\mathcal{A}$, α' satisfies P and π is an output of $\mathcal{A}$.

3. Serial Systems

In this paper we define two kinds of systems: "serial systems" and "generic systems". Serial systems describe serial execution of transactions. A serial system is defined for the purpose of giving a correctness condition for other systems, namely that the schedules of another system should look like schedules of the serial system to the transactions. As with serial executions of single-level transaction systems, serial systems are too inefficient to use in practice. Thus, we will define generic systems, which allow transactions to run concurrently or abort after performing some work; these systems use some algorithm at each object in order to cope with the demands of concurrency control and recovery.

In this section of the paper we define serial systems, which consist of *transaction automata* and *basic object automata* communicating with a *serial scheduler*. Transactions and basic objects describe user programs and data, respectively. The serial scheduler controls communication between the other components, and thereby controls the orders in which the transactions create children or access data. All the system components are modelled as I/O automata. Most of this section is taken from [LM], with slight modifications to accomodate minor changes in definitions.

We represent the pattern of transaction nesting by a *system type*, which is a set of transaction names, together with extra structure described below. We assume that each possible instantiation of a piece of program text has its own name. The transaction names are organized into a tree by the mapping "parent()", with T_0 as the root. In referring to this tree, which is part of the system type, we use traditional terminology, such as child, leaf, least common ancestor (lca), ancestor and descendant. (A transaction is its own ancestor and descendant.) Thus the children of a transaction name are the names of all the sub-transactions that might ever be created for it. The leaves of the tree of transaction names are called *accesses*. The accesses are partitioned, where each element of the partition contains the accesses to a particular object. We denote the element of the partition containing the accesses to object X by accesses(X). The partition, including the names of the objects, is part of the system type.

The system type can be thought of as a predefined naming scheme for all possible transaction instantiations that might ever be created, indicating for each instantiation the transaction of which it is a subtransaction, and also indicating (for those transactions that are accesses) the name of the object being accessed. In any particular execution, however, only some of these transactions will actually take steps. We assume that the tree structure is known in advance by all components of a system. The tree will, in general, be an infinite structure with infinite branching.

The root transaction T_0 plays a special role in this theory. The root models the environment of the nested transaction system (the "external world") from which requests for transactions originate and to which the results of these transactions are reported. Since it has no parent, T_0 may neither commit nor abort. The classical transactions of concurrency control theory (without nesting) appear in our model as the children of T_0. (In other work on nested transactions, such as Argus, the children of T_0 are often called "top-level" transactions.) Even in the context of classical theory (with no additional nesting) it is convenient to introduce the root transaction to model the environment in which the rest of the transaction system runs, with operations that describe the invocation and return of the classical transactions. It is natural to reason about T_0 in the same way as about all of the other transactions.

The only transactions that actually access data are the leaves of the transaction tree, and thus they are distinguished as "accesses". The internal nodes of the tree model transactions whose function is to create and manage subtransactions, but not to access data directly.

We also assume that a system type includes a designated set V of *values*, to be used as return values of transactions.

A serial system of a given system type is the composition of a set of I/O automata. This set contains a transaction automaton for each *internal* (i.e. non-leaf, non-access) node of the transaction tree, a basic object automaton for each object, and a serial scheduler. These automata are described below. Naturally, a practical system would not wish to have a separate process pre-existing for every possible instantiation of a piece of code, but would rather create processes dynamically as they are needed. For reasoning about the system, though, it is cleaner to model each instantiation separately as a permanent entity, with an operation to wake it up when it gets "created".

3.1. Transactions

This paper differs from other work such as [BBG] in that we model the transactions explicitly. A non-access *transaction* T is modelled as an I/O automaton, with the following operations.

Input operations:
  CREATE(T)
  REPORT_COMMIT(T',v), for T' a child of T, and v a value
  REPORT_ABORT(T'), for T' a child of T
Output operations:
  REQUEST_CREATE(T'), for T' a child of T
  REQUEST_COMMIT(T,v), for v a value

The CREATE input operation "wakes up" the transaction. The REQUEST_CREATE output operation is a request by T to create a particular child transaction.[6] The REPORT_COMMIT input operation reports to T the successful completion of one of its children, and returns a value recording the results of that child's execution. The REPORT_ABORT input operation reports to T the unsuccessful completion of one of its children, without returning any other information. We call REPORT_COMMIT(T',v), for any v, and REPORT_ABORT(T') *report* operations for transaction T'. The REQUEST_COMMIT operation is an announcement by T that it has finished its work, and includes a value recording the results of that work.

[6] Note that there is no provision for T to pass information to its child in this request. In a programming language, T might be permitted to pass parameter values to a subtransaction. Although this may be a convenient descriptive aid, it is not necessary to include it in the underlying formal model. Instead, we consider transactions that have different input parameters to be different transactions.

It is important to note that we use two separate operations, REQUEST_CREATE and CREATE, to describe what takes place when a subtransaction is activated. The REQUEST_CREATE is an operation of the transaction's parent, while the actual CREATE takes place at the subtransaction itself. In actual systems such as Argus, this separation does occur, and the distinction will be important in our results and proofs. Similarly, we distinguish between a subtransaction's REQUEST_COMMIT, the actual COMMIT (which is internal to the controller - see Section 3.3), and the REPORT_COMMIT operation of the parent transaction.[7]

We leave the details of particular transaction automata largely unspecified; in each system the choice of an automaton, that is the choice of which children to create, and what value to return, will depend on the particular piece of user code being modeled. For the purposes of the controllers studied here, the transactions (and in large part, the objects) are "black boxes." Nevertheless, it is convenient to assume that schedules of transaction automata obey certain syntactic constraints. We therefore require that all transaction automata preserve well-formedness, as defined in the next paragraph. We do not constrain how transaction automata behave once well-formedness has been violated, but we will prove later that a transactions generate only well-formed schedules when placed in any of the systems we consider.

We recursively define[8] *well-formedness* for sequences of operations of transaction T. Namely, the empty schedule is well-formed. Also, if $\alpha = \alpha'\pi$ is a sequence of operations of T, where π is a single event, then α is well-formed provided that α' is well-formed, and the following hold.

- If π is CREATE(T), then
  (i) there is no CREATE(T) event in α'.

- If π is REPORT_COMMIT(T',v) for a child T' of T, then
  (i) REQUEST_CREATE(T') appears in α' and
  (ii) there is no REPORT_ABORT(T') event in α' and
  (iii) there is no REPORT_COMMIT(T',v') event with $v' \neq v$ in α'.

- If π is REPORT_ABORT(T') for a child T' of T, then
  (i) REQUEST_CREATE(T') appears in α' and
  (ii) there is no REPORT_COMMIT event for T' in α'.

- If π is REQUEST_CREATE(T') for a child T' of T, then
  (i) there is no REQUEST_CREATE(T') event in α' and
  (ii) there is no REQUEST_COMMIT event for T in α' and

[7] Note that we do not include a REQUEST_ABORT operation for a transaction: we do not model the situation in which a transaction decides that its own existence is a mistake. Rather, we assign decisions to abort transactions to another component of the system, the controller. In practice, the controller must have some power to decide to abort transactions, as when it detects deadlocks or failures. In Argus, transactions are permitted to request to abort; we regard this request simply as a "hint" to the controller, to restrict its allowable executions in a particular way.

[8] This definition is more restrictive than that in [LM], where a REQUEST_COMMIT was allowed whenever the transaction had been created and had not requested to commit.

(iii) CREATE(T) appears in α'.

- If π is REQUEST_COMMIT(T,v) for a value v, then
  (i) there is no REQUEST_COMMIT event for T in α' and
  (ii) CREATE(T) appears in α' and
  (iii) there is a report event in α' for every child of T for which there is a REQUEST_CREATE event in α'.

These restrictions are very basic; they simply say that a transaction does not get created more than once, does not receive conflicting information about the fates of its children, and does not receive information about the fate of any child whose creation it has not requested; also, a transaction does not perform any output operations before it has been created or after it has requested to commit, it does not request the creation of the same child more than once, and it does not request to commit until it has received information about the fate of every child whose creation it requested. Except for these minimal conditions, there are no a priori restrictions on allowable transaction behavior.

The following easy lemma summarizes the properties of well-formed sequences of transaction operations.

Lemma 2: Let α be a well-formed sequence of operations of transaction T. Then the following conditions hold.

1. The first event in α is a CREATE(T) event, and there are no other CREATE events.
2. There is at most one REQUEST_CREATE(T') event in α for each child T' of T.
3. There are not two different report operations in α for any child T' of T. (However, there may be several events that are repeated instances of a single report operation).
4. Any report event for a child T' of T is preceded by REQUEST_CREATE(T') in α.
5. If a REQUEST_COMMIT event for T occurs in α, then there are no later REQUEST_CREATE or REQUEST_COMMIT events of T in α.
6. If a REQUEST_COMMIT event for T occurs in α, then it is preceded by a report event for each child of T whose creation is requested in α.

Conversely, any sequence of operations of T satisfying these conditions is well-formed.

3.2. Basic Objects

Recall that I/O automata are associated with non-access transactions only. Since access transactions model abstract operations on shared data objects, we associate a single I/O automaton with each object, rather than one for each access. The operations for each object are just the CREATE and REQUEST_COMMIT operations for all the corresponding access transactions. Although we give these operations the same sorts of names as the operations of non-access transactions, it is helpful to think of the operations of access transactions in other terms also: a CREATE corresponds to an invocation of an operation on the object, while a REQUEST_COMMIT corresponds to a response by the object to an invocation. One should notice though that these CREATE and REQUEST_COMMIT operations carry with them a designation of the position of the access in the transaction tree. Thus, a *basic object* X is modelled as an automaton, with the following operations.

Input operations:

CREATE(T), for T an access to X
Output operations:
REQUEST_COMMIT(T,v), for T an access to X

Our model differs significantly from other common models used to reason about data, in that we do not insist that the object have a value (of the type returned by a read operation) at all times. We do not specify any particular relationship between the internal state of the object automaton and the values returned by accesses, and it is only the values returned that matter to transactions in the system.

As with transactions, while specific objects are left largely unspecified, it is convenient to require that schedules of basic objects satisfy certain syntactic conditions. Thus each basic object is required to preserve well-formedness, as defined below.

Let α be a sequence of events of a basic object. Then an access T to X is said to be *pending* in α provided that α contains CREATE(T) but no REQUEST_COMMIT event for T. We recursively define *well-formedness* for sequences of operations of basic objects. Namely, the empty schedule is well-formed. Also, if $\alpha = \alpha'\pi$ is a sequence of operations of basic object X, where π is a single event, then α is well-formed provided that α' is well-formed, and the following hold.

- If π is CREATE(T), then
  (i) there is no CREATE(T) event in α', and
  (ii) there are no pending accesses in α'.

- If π is REQUEST_COMMIT(T,v) for a value v, then
  (i) there is no REQUEST_COMMIT event for T in α', and
  (ii) CREATE(T) appears in α'.

These restrictions simply say that the same access does not get created more than once, nor does the creation of a new access occur before any previous access has completed (i.e. requested to commit); also, a basic object does not respond more than once to any access, and only responds to accesses that have previously been created.

The following easy lemma summarizes the properties of well-formed sequences of basic object operations.

Lemma 3: Let α be a well-formed sequence of operations of basic object X. If α contains an even number of events, then α is the concatenation of a sequence of pairs CREATE(T_i)REQUEST_COMMIT(T_i,v_i), with $T_i \neq T_j$ when $i \neq j$. If α contains an odd number of operations, then α is the concatenation of a sequence of pairs CREATE(T_i)REQUEST_COMMIT(T_i,v_i) followed by a single CREATE(T'), with $T_i \neq T_j$ when $i \neq j$, and $T' \neq T_i$. Conversely any sequence α of operations of X that satisfies either of these descriptions is well-formed.

We will often have occasion to refer to a pair of operations CREATE(T)REQUEST_COMMIT(T,v).

REPORT_COMMIT(T,v), $T \neq T_0$
Precondition:
    $T \in$ s'.committed
    $(T,v) \in$ s'.commit_requested

The input operations, REQUEST_CREATE and REQUEST_COMMIT, simply result in the request being recorded. A CREATE operation can occur only if a corresponding REQUEST_CREATE has occurred and the CREATE has not already occurred. The second precondition on the CREATE operation says that the serial scheduler does not create a transaction until all its previously created sibling transactions have returned. That is, siblings are run sequentially. The preconditions on the ABORT operation say that the serial scheduler does not abort a transaction while any of its siblings are active or if the transaction has already been created. That is, aborted transactions take no steps and are dealt with sequentially with respect to their siblings. The result of a transaction can be reported to its parent at any time after the (purely internal) commit or abort has occurred. In particular, siblings might run in one order and be reported to their parent in some other order.[9]

The next lemma relates a schedule of the serial scheduler to the state that results from applying that schedule.

Lemma 4: Let α be a schedule of the serial scheduler, and let s be a state that can result from applying α to the initial state. Then the following conditions are true.

1. T is in s.create_requested exactly if $T = T_0$ or α contains a REQUEST_CREATE(T) event.
2. T is in s.created exactly if α contains a CREATE(T) event.
3. (T,v) is in s.commit_requested exactly if α contains a REQUEST_COMMIT(T,v) event.
4. T is in s.committed exactly if α contains a COMMIT(T) event.
5. T is in s.aborted exactly if α contains an ABORT(T) event.
6. s.returned = s.committed $\cup$ s.aborted.
7. s.committed $\cap$ s.aborted = $\emptyset$.

3.4. Serial Systems and Serial Schedules

The composition of transactions with basic objects and the serial scheduler for a given system type is called a *serial system*, and its operations and schedules are called *serial operations* and *serial schedules*, respectively. We note that every serial operation is an operation of the serial scheduler and of at most one other component of the serial system. A sequence α of serial operations is said to be *well-formed* provided that its projection at every transaction and basic object is well-formed.

[9] One significant difference between our serial scheduler and the one in [LM] is that there the return operation and the report to the parent of the return are combined as a single operation, giving the parent the extra information of the order in which its children are run. Another difference is that in [LM] the serial scheduler prevented a transaction from committing unless every child whose creation was requested had returned, since in that paper the transactions were not required to enforce this themselves.

If α is a sequence of operations and T is a transaction such that α contains CREATE(T) but no return event for T, we say that T is *live* in α. The following are useful observations:

Lemma 5: Let α be a well-formed serial schedule, T a transaction live in α and T' an ancestor of T. Then T' is live in α.

Lemma 6: Let α be a well-formed serial schedule, and T and T' distinct sibling transactions. If T is live in α, then T' is not live in α.

A consequence of the two previous lemmas is the following, which states that transactions can be live concurrently in a well-formed serial schedule only if one is an ancestor of the other.

Lemma 7: Let α be a well-formed serial schedule, and T and T' transactions each of which is live in α. Then either T is an ancestor of T' or T' is an ancestor of T.

We now show that all serial schedules are well-formed. Afterwards, we will use this fact repeatedly, without explicitly noting it or referencing this lemma.

Lemma 8: Let α be a serial schedule. Then α is well-formed.

Proof: By induction on the length of schedules. The base, length = 0, is trivial. Suppose that $\alpha\pi$ is a serial schedule, and assume that α is well-formed. If π is an output of a basic object or non-access transaction P, then $\alpha\pi|P$ is well-formed because P preserves well-formedness, and so $\alpha\pi$ is well-formed. So assume that π is an output operation of the serial scheduler. If π is a return operation for a transaction, it is not an operation of any basic object or non-access transaction automata, so $\alpha\pi$ is well-formed, since $\alpha\pi|P=\alpha|P$ for all basic object or non-access transaction automata P. The remaining cases are when π is an input operation of some basic object or non-access transaction automaton P. It suffices, in each case, to show that $\alpha\pi|P$ is well-formed.

(1) π is CREATE(T) for some non-access transaction T.
The serial scheduler preconditions and Lemma 4 ensure that CREATE(T) does not appear in α.

(2) π is CREATE(T) for some access T to basic object X.
The serial scheduler preconditions and Lemma 4 ensure that CREATE(T) does not appear in α. Well-formedness of α at X implies therefore that α does not contain a REQUEST_COMMIT event for T, and thus that T is pending in α. By Lemma 7 we deduce that no transactions except ancestors of T are live in α, and in particular, no other access to X is live in α. Any pending access T' in α must be live, as α cannot contain a COMMIT event for T' without also containing a REQUEST_COMMIT event for T', and α cannot contain an ABORT event for T as well as a CREATE event for T'. Thus no access to X other than T is pending in α.

(3) π is REPORT_COMMIT(T,v) for some transaction T and value v.
Then π is an input to transaction parent(T) = T'. The serial scheduler preconditions and Lemma 4 imply that α contains REQUEST_COMMIT(T,v). Well-formedness of α at the non-access transaction T (or at basic object X, if T is an access to X) implies that α contains CREATE(T), and the serial scheduler preconditions and Lemma 4 then require that α contains REQUEST_CREATE(T). Also, serial scheduler preconditions and Lemma 4 imply that COMMIT(T) occurs in α, and thus no ABORT(T) occurs in α. Thus no REPORT_ABORT(T) occurs in α, by the serial scheduler preconditions. Well-formedness at T (or at X, if T is an access to X) implies that no REQUEST_COMMIT(T,v') with v' $\neq$ v occurs in α, and therefore by the serial scheduler preconditions, no REPORT_COMMIT(T,v')

with v' $\neq$ v occurs in α.

(4) π is REPORT_ABORT(T) *for some transaction* T.
Then π is an input to transaction parent(T) = T'. The serial scheduler preconditions and Lemma 4 imply that α contains ABORT(T) and hence contains REQUEST_CREATE(T) but no CREATE(T). The analysis above shows that this is incompatible with the presence of any REPORT_COMMIT event for T. □

3.5. Visibility

In order to talk about schedules, we introduce some terms to describe the fate of transactions. Let α be any sequence of operations. (We will use these same terms later for schedules of generic systems, so we make the definitions for general sequences.) If T is a transaction and T' an ancestor of T, we say that T is *committed* to T' in α if COMMIT(U) occurs in α for every U that is an ancestor of T and a proper descendant of T'. If T and T' are transactions we say that T is *visible* to T' in α if T is committed to lca(T,T').

We also introduce two terms that describe different relationships between events and transactions. We will always associate an operation of a non-acess transaction automaton with that transaction. The return operation for a transaction (which occurs only at the controller) will sometimes need to be associated with the transaction, and sometimes with the parent of the transaction. If π is one of the operations CREATE(T), REQUEST_CREATE(T'), REPORT_COMMIT(T',v'), REPORT_ABORT(T',v'), REQUEST_COMMIT(T,v), COMMIT(T), or ABORT(T), where T' is a child of T, we say that π *mentions* T. Thus if T is a non-access transaction then the operations that mention T are the operations of the automaton T together with the return operations for T. If π is one of the operations CREATE(T), REQUEST_CREATE(T'), COMMIT(T'), ABORT(T'), REPORT_COMMIT(T',v'), REPORT_ABORT(T',v'), or REQUEST_COMMIT(T,v), where T' is a child of T, then we define *transaction(π)* to be T. If T is a non-access transaction then the operations π with transaction(π) = T are the operations of the automaton T together with the return operations for children of T. We denote by visible(α,T) the subsequence of α consisting of events π with transaction(π) visible to T in α. Notice that every operation occurring in visible(α,T) is a serial operation, even if α itself contains other operations.

We collect here some straightforward consequences of these definitions:

Lemma 9: Let α be a sequence of operations, and T, T' and T'' transactions.

1. If T is an ancestor of T', then T is visible to T' in α.
2. T' is visible to T in α if and only if T' is visible to lca(T,T'') in α.
3. If T'' is visible to T' in α and T' is visible to T in α, then T'' is visible to T in α.
4. If T' is a proper descendant of T, T'' is visible to T' in α, but T'' is not visible to T in α, then T'' is a descendant of the child of T that is an ancestor of T'.

Lemma 10: Let α and β be sequences of operations such that β consists of a subset of the events of α.

1. If transaction T is visible to transaction T' in β, then T is visible to T' in α.

2. If event π is in visible(β,T), then π is in visible(α,T).

Lemma 11: Let α be a sequence of operations, and let T and T' be transactions. Then visible(α,T)|T' is equal to α|T' if T' is visible to T in α, and is equal to the empty sequence otherwise.

Lemma 12: Let α be a sequence of operations. Let T, T' and T'' be transactions such that T'' is visible to T' and to T in α. Then T'' is visible to T' in visible(α,T).

Lemma 13: Let T be a transaction, and let $\alpha\pi$ be a sequence of operations, where π is a single event.

1. If transaction(π) is not visible to T in $\alpha\pi$, then visible($\alpha\pi$,T) = visible(α,T).
2. If transaction(π) is visible to T in $\alpha\pi$ and if π is not a COMMIT event, then visible($\alpha\pi$,T) = visible(α,T)π.
3. If transaction(π) is visible to T in $\alpha\pi$, and π is COMMIT(U) then the events in visible($\alpha\pi$,T) are those visible in α to either T or U, together with π itself.

Let α be any sequence of operations. If T is a transaction we say T is an *orphan* in α if ABORT(U) occurs in α for some ancestor U of T. The following lemmas are straightforward.

Lemma 14: Let α and β be sequences of operations such that β consists of a subset of the events in α. If a transaction T is not an orphan in α then T is not an orphan in β.

Lemma 15: Let α is a sequence of operations. If T is a transaction that is not an orphan in α and T' is an ancestor of T, then T' is not an orphan in α.

3.6. Serial Correctness

We use serial schedules as the basis of our correctness definition, which was first given in [LM]. Namely, we say that a sequence of operations is *serially correct for a transaction T* provided that its projection on T is identical to the projection on T of some serial schedule. That is, the sequence "looks like" a serial schedule to T. Later in this paper we will define "Conflict-Based Locking systems" and show that their schedules are serially correct for every non-orphan transaction, and in particular that these schedules are serially correct for the root transaction T_0.

Motivation for our use of serial schedules to define correctness derives from the simple behavior of the serial scheduler, which determines the sequence of interactions between the transactions and objects. We believe the depth-first traversal of the transaction tree to be a natural notion of correctness that corresponds precisely to the intuition of how nested transaction systems ought to behave. Furthermore, it is a natural generalization of serializability, the correctness condition generally chosen for classical transaction systems. Serial correctness for T is a condition that guarantees to implementors of T that their code will encounter only situations that can arise in serial executions. Correctness for T_0 is a special case that guarantees that the external world will encounter only situations that can arise in serial executions.

It would be best if every transaction (whether an orphan or not) saw data consistent with a serial execution. Ensuring this requires a much more intricate controller than the simple generic systems we describe below. In [HLMW], several algorithms for maintaining correctness for orphan transactions are described and verified.

Our approach is an example of a general technique for studying system algorithms. A simple, intuitive but inefficient or impractical algorithm (automaton) is used to specify an acceptable collection of schedules for the system component. The actual system component is more efficient or robust, but provides the same user interface. The user is guaranteed that applications (transactions, in our work) that work well when run with the simple algorithm will work the same way when run with the actual system.

4. Generic Systems

The serial systems described in the previous section have a restrictive controller (the serial scheduler) that makes their behavior easy to reason about, but also makes them a poor choice for implementation. Practical systems allow activity to procede concurrently in sibling transactions to improve performance, and therefore need to use complicated algorithms at the objects to ensure that the resulting behavior is still serially correct. In this section, we introduce a generic system, which is composed of transactions (just like a serial system), generic objects (each of which receives information about commits and aborts of transactions, and uses an as yet unspecified concurrency control algorithm), and a generic controller (which allows concurrency and aborts of running transactions). In Theorem 32, we will give a simple condition on the generic objects that ensures that the generic system is serially correct.

4.1. Generic Objects

For each object X in the system type, a generic system must include a generic object automaton. The operations of a generic object X are:

Input operations:
CREATE(T), T an access to X
INFORM_COMMIT_AT(X)OF(T)
INFORM_ABORT_AT(X)OF(T)
Output operations:
REQUEST_COMMIT(T,v), T an access to X

We give a recursive definition for *well-formedness* of schedules of generic object X. Namely, the empty schedule is well-formed. Also, if $\alpha = \alpha'\pi$ is a sequence of events of generic object X, then α is well-formed provided that α' is well-formed and the following hold.

- If π is CREATE(T), then
  (i) there is no CREATE(T) in α'.

- If π is a REQUEST_COMMIT for T, then
  (i) there is no REQUEST_COMMIT for T in α', and

(ii) CREATE(T) occurs in α'.

- If π is INFORM_COMMIT_AT(X)OF(T), then
  (i) there is no INFORM_ABORT_AT(X)OF(T) in α', and
  (ii) if T is an access to X, then a REQUEST_COMMIT for T occurs in α'.

- If π is INFORM_ABORT_AT(X)OF(T), then
  (i) there is no INFORM_COMMIT_AT(X)OF(T) in α'.

Generic objects are required to preserve well-formedness.

Each generic object represents a data item, together with the concurrency control and recovery algorithms used to maintain it. Thus our model is particularly convenient for representing systems in which different data items are maintained with different concurrency control algorithms, as can easily happen in a distributed database formed by combining pre-existing databases. It is well known that different correct algorithms cannot always be combined. For example, a system, in which some data items are maintained with timestamps and others use two-phase locking, may not guarantee serializability. In this paper we will introduce a property called dynamic atomicity. We will show that so long as every generic object in the system is dynamic atomic, then the whole system generates serially correct schedules.

4.2. Generic Controller

The generic controller is a nondeterministic automaton. It passes requests for the creation of subtransactions or accesses to the appropriate recipient, passes responses back to the caller and informs objects of the fate of transactions, but it may delay such messages for arbitrary lengths of time or unilaterally decide to abort a subtransaction that has been created. Moss [Mo] devotes considerable effort to describing a distributed implementation of the controller that copes with communication failures and loss of system information due to crashes, yet still commits a subtransaction whenever possible. These concerns are orthogonal to the correctness of the data management algorithms and we do not address them here.[10] We use nondeterminism in the generic controller so that our results will apply to many different controllers, each implementing the generic controller by exhibiting a subset of the behaviors permitted by the nondeterminism.

The generic controller has the following operations:

Input operations:
REQUEST_CREATE(T)
REQUEST_COMMIT(T,v)
Output operations:

[10] The generic controller is similar to the weak concurrent controller of [LM]. It differs slightly in the names of its operations, in the separation of return and report operations, and in the conditions under which CREATE and COMMIT operations are permitted to occur.

CREATE(T)
COMMIT(T), $T \neq T_0$
ABORT(T), $T \neq T_0$
REPORT_COMMIT(T,v), $T \neq T_0$
REPORT_ABORT(T), $T \neq T_0$
INFORM_COMMIT_AT(X)OF(T), $T \neq T_0$
INFORM_ABORT_AT(X)OF(T), $T \neq T_0$

These play the same roles as in the serial scheduler, with the addition of the INFORM_COMMIT and INFORM_ABORT operations, which pass information about the fate of transactions to the generic objects.

Each state s of the generic controller consists of six sets: s.create_requested, s.created, s.commit_requested, s.committed, s.aborted and s.returned. The set s.commit_requested is a set of (transaction,value) pairs, and the others are sets of transactions. All are empty in the initial state except for create_requested, which is $\{T_0\}$.

The operations are defined by pre- and postconditions as follows:

REQUEST_CREATE(T)
Postcondition:
s.create_requested = s'.create_requested $\cup$ {T}

REQUEST_COMMIT(T,v)
Postcondition:
s.commit_requested = s'.commit_requested $\cup$ {(T,v)}

CREATE(T), T a transaction
Precondition:
T $\in$ s'.create_requested - s'.created
Postcondition:
s.created = s'.created $\cup$ {T}

COMMIT(T), $T \neq T_0$
Precondition:
(T,v) $\in$ s'.commit_requested for some v
T $\notin$ s'.returned
Postcondition:
s.committed = s'.committed $\cup$ {T}
s.returned = s'.returned $\cup$ {T}

ABORT(T), $T \neq T_0$
Precondition:
T $\in$ s'.create-requested - s'.returned
Postcondition:
s.aborted = s'.aborted $\cup$ {T}
s.returned = s'.returned $\cup$ {T}

REPORT_COMMIT(T,v), $T \neq T_0$
Precondition:
$T \in$ s'.committed
$(T,v) \in$ s'.commit_requested

REPORT_ABORT(T), $T \neq T_0$
Precondition:
$T \in$ s'.aborted

INFORM_COMMIT_AT(X)OF(T), $T \neq T_0$
Precondition:
$T \in$ s'.committed

INFORM_ABORT_AT(X)OF(T), $T \neq T_0$
Precondition:
$T \in$ s'.aborted

Lemma 16: Let α be a schedule of the generic controller, and let s be the state that results from applying α to the initial state s_0. Then the following conditions are true.

1. T is in s.create_requested exactly if $T = T_0$ or α contains a REQUEST_CREATE(T) event.

2. T is in s.created exactly if α contains a CREATE(T) event.

3. (T,v) is in s.commit_requested exactly if α contains a REQUEST_COMMIT(T,v) event.

4. T is in s.aborted exactly if α contains an ABORT(T) event.

5. T is in s.committed exactly if α contains a COMMIT(T) event.

6. s.returned = s.committed $\cup$ s.aborted.

7. s.committed $\cap$ s.aborted = $\emptyset$.

An obvious but important property of the generic controller is given by the next lemma.

Lemma 17: If α is a schedule of the generic controller then α contains at most one return event for each transaction T.

4.3. Generic Systems

A *generic system* is the composition of non-access transaction automata, generic object automata, and the generic controller automaton. The operations of a generic system are called *generic operations*, and a schedule of a generic system is called a *generic schedule*. A sequence of generic events is called *well-formed* provided that its projection at each non-access transaction and generic object is well-formed.

We collect here some straight-forward properties of generic schedules.

Lemma 18: Let α be a generic schedule. Then α is well-formed.

Lemma 19: Let α be a generic schedule, and T a transaction that is not an orphan in α. If T' is visible to T in α, then T' is not an orphan in α.

Lemma 20: Let α be a generic schedule, and T a transaction that is live and not an orphan in α. Then the ancestors of T are all live in α.

5. Reordering Events: Return Order and the Affects Order

The correctness condition we seek to demonstrate requires non-orphan transactions to see serial schedules; given a generic schedule α and a non-orphan transaction T, we wish to prove that $\alpha|T = \beta|T$ for some serial schedule β. To find β, our strategy will be to extract from α the subsequence visible(α,T), which contains all the operations of T, and then reorder the events in visible(α,T) to form the new sequence β, which we argue is a serial schedule and looks to T exactly like α provided the generic objects all obey a simple condition.

Serial schedules are characterised by the depth-first order in which the serial scheduler runs transactions. This depth-first order can be characterized by ordering each set of siblings. Below, we define such an ordering to be a "sibling order," and focus attention on the sibling order induced by the order of return events in the generic schedule α. As we will see, this is the order in which locking protocols serialize transactions. We use the return order on siblings to induce a partial order "$\mathrm{return}_E(\alpha)$" on the events of α. When we reorder the events of visible(α,T) to produce β, we will use the return order so that the transactions take steps in β in an appropriate depth-first order (as they must if β is to be a serial schedule).

When we are trying to produce a serial schedule by reordering the operations of visible(α,T), we must be careful not to introduce absurdities, such as moving a CREATE event before the corresponding REQUEST_CREATE. We introduce a relation on the events in α, "$\mathrm{affects}_E(\alpha)$", which records the possible causal relation between different events.

Two relations R and S are *consistent* if their union can be extended to an irreflexive partial order. (That is, their union has no cycles.) In this section of the paper, we show that $\mathrm{return}_E(\alpha)$ and $\mathrm{affects}_E(\alpha)$ are consistent orderings on the events of α. We use an extension of the union of these orders when reordering visible(α,T) to form β. We will argue in the next section of the paper that if the generic objects obey a simple condition called dynamic atomicity, then the reordered sequence β is a schedule of the serial system. Later we will show that objects implemented using our Conflict-Based Locking algorithm or Moss' Read/Write Locking algorithm are dynamic atomic.

5.1. Return Order

Let *SIB* be the (irreflexive) sibling relation among transactions, so (T,T') $\in$ SIB if and only if T $\neq$ T' and parent(T)=parent(T'). If R $\subseteq$ SIB is a strict partial order (i.e. a relation that is transitive, irreflexive and anti-symmetric) then we call R a ***sibling order***. If α is a sequence of events, then we define return(α) to be the binary relation on transactions containing (T,T') exactly if T and T' are siblings and one of the following holds.

- There are return events for both T and T' in α, and a return event for T precedes a return event for T'.
- There is a return event for T in α, but there is no return event for T' in α.

Lemma 21: Let α be a generic schedule. Then return(α) is a sibling order.

We will now use the return order on siblings to induce return orders on transactions, events and deeds. The partial order on events will be the one we use in reordering visible(α,T) to give a serial schedule. If α is a sequence of events, we define $\text{return}_T(\alpha)$ to be the binary relation on transactions containing (T,T') exactly when there exist siblings U and U' such that T and T' are descendants of U and U', respectively, and (U,U') $\in$ return(α). We also define $\text{return}_E(\alpha)$ to be the binary relation on the events of α containing (ϕ,π) exactly when ϕ and π are distinct events of α mentioning transactions T and T' respectively, such that (T,T') $\in$ $\text{return}_T(\alpha)$. Similarly, we define $\text{return}_D(\alpha)$ to be the binary relation on deeds containing ((T,t)(T',t')) exactly when α contains events π and ϕ such that π=REQUEST_COMMIT(T,t), ϕ=REQUEST_COMMIT(T',t'), and (π,ϕ) $\in$ $\text{return}_E(\alpha)$. Clearly, each of these derived relations is a strict partial order.

Lemma 22: Let α be a generic schedule. Then $\text{return}_T(\alpha)$ is a strict partial order on transactions, $\text{return}_E(\alpha)$ is a strict partial order on the set of events of α, and $\text{return}_D(\alpha)$ is a strict partial order on deeds.

Lemma 23: Let α be a generic schedule. Let π and π' be events of α mentioning transactions T and T' respectively. Let ψ and ψ' be events of α mentioning transactions U and U' respectively, where U is a descendant of T and U' is a descendant of T'. If (π,π') $\in$ $\text{return}_E(\alpha)$ then (ψ,ψ') $\in$ $\text{return}_E(\alpha)$.

5.2. Affects Order

We now define another partial order $\text{affects}_E(\alpha)$ on the events of α. This will record those facts about the neccessary order of events that must be enforced by the preconditions of the serial scheduler. For a sequence α of events, and events ϕ and π in α, we say that ϕ ***directly affects*** π in α (and that (ϕ,π) $\in$ directly-$\text{affects}_E(\alpha)$) if at least one of the following is true.

- ϕ and π are distinct events of the same transaction (including accesses)and ϕ precedes π in α
- ϕ = REQUEST_CREATE(T) and π = CREATE(T)
- ϕ = REQUEST_COMMIT(T,v) and π = COMMIT(T)

- ϕ = REQUEST_CREATE(T) and π = ABORT(T)
- ϕ = COMMIT(T) and π = REPORT_COMMIT(T,v)
- ϕ = ABORT(T) and π = REPORT_ABORT(T)

Examining the definition of the serial scheduler, we see that $(\phi,\pi) \in$ directly-affects$_E(\alpha)$ either when the preconditions of π as an operation of the serial scheduler include a test for a previous occurrence of ϕ, in which case a sequence of operations with π not preceded by ϕ could not possibly be a serial schedule, or when π and ϕ are operations of the same transaction, in which case we might risk introducing an absurdity if π was not preceded by ϕ, as we assume very little about the transaction automata. Since the generic controller preconditions also test for the presence of a preceding operation in each case and do not allow repeated instances of REQUEST_CREATE, REQUEST_COMMIT, or return operations, it is clear that if α is a generic schedule, then $(\phi,\pi) \in$ directly-affects$_E(\alpha)$ only if ϕ precedes π in α.

For a sequence α of events, define the relation affects$_E(\alpha)$ to be the transitive closure of the relation directly-affects$_E(\alpha)$. If the pair (ϕ,π) is in the relation affects$_E(\alpha)$, we also say that ϕ *affects* π in α.

Lemma 24: Let α be a generic schedule. Then affects$_E(\alpha)$ is an irreflexive partial order on the events in α.

Proof: We noted above that ϕ directly affects π in α only if ϕ precedes π in α. Therefore ϕ affects π in α only if ϕ precedes π in α. Thus affects$_E(\alpha)$ is irreflexive and antisymmetric. Since affects$_E(\alpha)$ is constructed as a transitive closure, the result follows. □

We now show that, as claimed earlier, the return and affects orders are consistent partial orders on the set of events of a generic schedule. We begin with two lemmas.

Lemma 25: Let α be a generic schedule such that ϕ directly affects π in α, where ϕ mentions T and π mentions T'. Then T = T', T = parent(T') or T' = parent(T).

Lemma 26: Let α be a generic schedule. Let π and π' be distinct events in α mentioning transactions T and T' respectively. If T is neither an ancestor nor a descendant of T', and $(\pi,\pi') \in$ affects$_E(\alpha)$, then $(\pi,\pi') \in$ return$_E(\alpha)$.

Proof: If T is neither an ancestor nor a descendant of T', there must be siblings U and U' such that T is a descendant of U, and T' is a descendant of U'. Let U'' denote parent(U)=parent(U')=lca(T,T'). Since π affects π' in α, there must be a subsequence $\sigma=\pi_1 ... \pi_n$ of α such that $\pi_1=\pi$ and $\pi_n=\pi'$, and where each π_i directly affects π_{i+1}.

By Lemma 25, by replacing each event in σ with the transaction it mentions, σ corresponds to a path through the transaction tree, beginning at T and ending at T', where each link $\pi_i\pi_{i+1}$ is either a self-loop, goes from child to parent or from parent to child.

Clearly, there must be some step $\pi_i\pi_{i+1}$ from U to U'', and a later step $\pi_j\pi_{j+1}$ from U'' to U'. Examining the definition of directly_affects$_E$, we have that π_i is a return for U, and π_{i+1} is a report for U. Similarly, π_j is REQUEST_CREATE(U') and π_{j+1} is either CREATE(U') or ABORT(U'). The controller preconditions and well-formedness imply that any return event for U' in α must occur after the unique REQUEST_CREATE(U') event. Thus α contains a return event π_i for U, and any return event for U' must be preceded by the later event π_j. Thus (U,U') $\in$ return(α), and therefore (π,π') $\in$ return$_E$(α). □

Now we will prove that the two partial orders we have defined on the events of α are consistent.

Lemma 27: Let α be a generic schedule. Then return$_E$(α) and affects$_E$(α) are consistent partial orders on the events of α.

Proof: We prove this lemma by contradiction. If return$_E$(α) and affects$_E$(α) are not consistent, then there is a cycle in the relation return$_E$(α) $\cup$ affects$_E$(α), and thus there must be some shortest cycle. Let π_0, π_1, π_2, ..., π_{n-1}, π_n=π_0 be such a shortest cycle, where for each i, (π_i,π_{i+1}) $\in$ return$_E$(α) $\cup$ affects$_E$(α). In the following discussion we will use arithmetic modulo n for subscripts, so that if i=0, π_{i-1} is to be interpreted as π_{n-1}. We note that n>1, since both return$_E$(α) and affects$_E$(α) are irreflexive.

Since the relation return$_E$(α) is acyclic, there must be at least one index i such that (π_i,π_{i+1}) $\in$ affects$_E$(α). Let T and T' be the transactions mentioned by π_i and π_{i+1} respectively. We divide the discussion into three cases, depending on the relationship between T and T' in the transaction tree.

If T is an ancestor of T', then consider the pair (π_{i-1},π_i). If this pair is in affects$_E$(α), then by the transitivity of the affects relation, (π_{i-1},π_{i+1}) $\in$ affects$_E$(α). However if (π_{i-1},π_i) $\in$ return$_E$(α), then by Lemma 23 (π_{i-1},π_{i+1}) $\in$ return$_E$(α). In either situation, we can find a shorter cycle in the relation return$_E$(α) $\cup$ affects$_E$(α), by omitting π_i. This contradicts our assumption, since the cycle chosen was as short as possible.

If T is a descendant of T', we consider the pair (π_{i+1},π_{i+2}). If this pair is in affects$_E$(α), then by the transitivity of the affects relation, (π_i,π_{i+2}) $\in$ affects$_E$(α). However if (π_{i+1},π_{i+2}) $\in$ return$_E$(α), then by Lemma 23 (π_i,π_{i+2}) $\in$ return$_E$(α). In either situation, we can find a shorter cycle in the relation return$_E$(α) $\cup$ affects$_E$(α), by omitting π_{i+1}. This contradicts our assumption, since the cycle chosen was as short as possible.

Finally, if T is neither an ancestor nor a descendant of T', then by Lemma 26, (π_i,π_{i+1}) is in return$_E$(α) as well as in affects$_E$(α). Now consider the pair (π_{i-1},π_i). If this pair is in affects$_E$(α), then by the transitivity of the affects relation, (π_{i-1},π_{i+1}) $\in$ affects$_E$(α). However if (π_{i-1},π_i) $\in$ return$_E$(α), then by the transitivity of the return relation (π_{i-1},π_{i+1}) $\in$ return$_E$(α). In either situation, we can find a shorter cycle in the relation return$_E$(α) $\cup$ affects$_E$(α), by omitting π_i. This contradicts our assumption, since the cycle chosen was as short as possible.

In every case we have found a contradiction, thus the assumption that the relation return$_E$(α) $\cup$ affects$_E$(α) contains a cycle must be wrong. □

5.3. Visibility

Recall that visible(α,T) is the subsequence of α consisting of the events π such that transaction(π) is visible to T. We begin our study of the way the return and affects orders relate the events of visible(α,T), in preparation for the main correctness proof of the next section.

First we show that two transactions mentioned in such a subsequence must be ordered by the return order, unless one is an ancestor of the other, and that therefore the return order determines the relative order of many of the events in the subsequence.

Lemma 28: Let α be a generic schedule, and T'' any transaction. Let π and π' be events in visible(α,T'') such that there are transactions T and T' where π mentions T, π' mentions T', and T is neither an ancestor nor a descendant of T'. Then either (π,π') $\in$ $\text{return}_E(\alpha)$, or (π',π) $\in$ $\text{return}_E(\alpha)$.

Proof: Let U and U' be distinct sibling transactions such that T is a descendant of U, and T' is a descendant of U'. Since U and U' are distinct siblings, T'' is not a descendant of both U and U'. Without loss of generality, we will assume that T'' is not a descendant of U. Note that therefore the least common ancestor of T'' and a descendant of U must be an ancestor of parent(U). Since π mentions T, transaction(π) must be T or parent(T). Thus transaction(π) is a descendant of U unless π is a return event for U itself. Since π is visible to T'' in α, α must contain a COMMIT event for every transaction that is an ancestor of transaction(π) and a strict descendant of lca(transaction(π),T''). If transaction(π) is a descendant of U, then α must therefore contain a COMMIT(U) event. If transaction(π) is not a descendant of U, then we saw that π itself must be a return event for U. In either case α contains a return event for U, and so return(α) orders U and U'. Therefore $\text{return}_T(\alpha)$ orders T and T'. An immediate consequence is that $\text{return}_E(\alpha)$ orders π and π'. □

Lemma 29: Let α be a generic schedule, let T be any transaction and X any object. Then there is a unique reordering of visible(α,T)|X consistent with $\text{return}_E(\alpha) \cup \text{affects}_E(\alpha)$.

Proof: By Lemma 27 the partial orders $\text{return}_E(\alpha)$ and $\text{affects}_E(\alpha)$ are consistent. This implies the existence of a reordering of visible(α,T)|X consistent with both. Uniqueness of the reordering will be proved once we show that any two events of visible(α,T)|X are ordered by one of the partial orders. Thus, let π and π' be distinct events of visible(α,T)|X. Let U and U', respectively, be the accesses of which π and π' are operations. If U=U', then π and π' are ordered by $\text{affects}_E(\alpha)$. If U $\neq$ U', then, since U and U' are accesses, U is neither an ancestor nor a descendant of U'. By Lemma 28, $\text{return}_E(\alpha)$ orders π and π'. □

For α a generic schedule, T a transaction and X an object, define *serialize(α,T,X)* to be the unique sequence that is a reordering of visible(α,T)|X consistent with $\text{return}_E(\alpha) \cup \text{affects}_E(\alpha)$, which exists by Lemma 29.

A useful observation is the following:

Lemma 30: Let α be a generic schedule, T a transaction that is not an orphan in α, and X an object. The sequence serialize(α,T,X) is well-formed as a sequence of operations of basic object X.

Proof: Suppose π is an event of serialize(α,T,X). There are two cases to consider: either π is CREATE(U) or π is REQUEST_COMMIT(U,u).

If π is CREATE(U), then since $\alpha|X$ is a well-formed schedule of generic object X, α contains only one CREATE(U) event. Thus serialize(α,T,X) contains no CREATE(U) event other than π. We also need to prove that no access to X is pending in the prefix of serialize(α,T,X) preceding π. Thus suppose that U' is an access to X such that CREATE(U') precedes π in serialize(α,T,X). Then U' $\neq$ U, and since visible(α,T) contains both CREATE(U') and π, by Lemma 28 we deduce that (CREATE(U'),π) $\in$ $\text{return}_E(\alpha)$. Therefore (U',U) $\in$ $\text{return}_T(\alpha)$. Thus α contains a return event for some ancestor of U', and since CREATE(U') is visible to T in α, U' is not an orphan in α. Thus by Lemma 20 α contains a COMMIT(U') event, and therefore also a REQUEST_COMMIT(U',u') event for some value u'. Since (U',U) $\in$ $\text{return}_T(\alpha)$, we deduce that (REQUEST_COMMIT(U',u'),π) $\in$ $\text{return}_E(\alpha)$, so that REQUEST_COMMIT(U',u') precedes π in serialize(α,T,X). Thus U' is not pending in the prefix of serialize(α,T,X) preceding π.

If π is REQUEST_COMMIT(U,u), then since $\alpha|X$ is a well-formed schedule of generic object X, α contains CREATE(U) preceding π, and α contains no REQUEST_COMMIT event for U other than π. Thus (CREATE(U),π) $\in$ $\text{affects}_E(\alpha)$ and since CREATE(U) is in visible(α,T), CREATE(U) precedes π in serialize(α,T,X). Also serialize(α,T,X) does not contain a REQUEST_COMMIT event for U other than π. □

6. Proving Serial Correctness

6.1. The Serial Correctness Theorem

The following lemma establishes conditions on generic objects that are sufficient to ensure the correctness of generic systems; i.e. that non-orphans see serial schedules. Actually, the statement and proof of the lemma establishes a stronger property: from any serial schedule α and non-orphan transaction T, we explicitly construct a serial schedule β that is *transaction equivalent* to visible(α,T), i.e. $\beta|T'$=visible(α,T)$|T'$ for all transactions T'. Note that if T' is visible to T in α, then by Lemma 13 visible(α,T)$|T' = \alpha|T'$. Thus, the single serial schedule β is a serial schedule that is consistent with the local schedules of every transaction visible to T.

Lemma 31: Let α be a generic schedule, and let T be any transaction that is not an orphan in α. Suppose that for each object X, serialize(α,T,X) is a schedule of basic object X. Then there is a serial schedule β such that β and visible(α,T) are transaction-equivalent.

Proof: Note first that by Lemma 29, serialize(α,T,X) is well defined.

By Lemma 27, $\text{affects}_E(\alpha) \cup \text{return}(\alpha)_E$ is an acyclic relation on the events of α, and hence on the events of visible(α,T). Then there exist total orderings on the events of visible(α,T) that extend $\text{affects}_E(\alpha) \cup \text{return}(\alpha)_E$. Let β be a sequence obtained by reordering visible(α,T) by such a total ordering. We argue that β and visible(α,T) are transaction-equivalent, and that β is a serial schedule.

First we note that since β is a rearrangement of visible(α,T), the events in $\beta|T'$ are the same as the events in visible(α,T)$|T'$, for any T'. Suppose π and ϕ are two events of visible(α,T)$|T'$, and π precedes ϕ in visible(α,T). Then (π,ϕ) $\in$ $\text{directly-affects}_E(\alpha)$, and so π must precede ϕ in β, since the order of events in β is consistent with $\text{affects}_E(\alpha)$. This shows that $\beta|T'$=visible(α,T)$|T'$. Thus β is transaction-equivalent to visible(α,T).

Now we argue that β is a serial schedule. The proof is by induction on prefixes of β, with a trivial basis. Let $\beta'\pi$ be a prefix of β with π a single event, assume that β' is serial, and let s' be the state of the serial scheduler after β'. There are six cases.

1) π is an output of a non-access transaction T'.
Then T' must be visible to T in α, and so by Lemma 13 visible(α,T)|T'=α|T'. We saw above that β and visible(α,T) are transaction equivalent, so we may deduce that β|T'=α|T'. Thus $\beta'\pi$|T' (which is a prefix of β|T') is a prefix of α|T', which is a schedule of T'. Thus $\beta'\pi$|T' is a schedule of T'. The result follows by Lemma 1.

2) π is an output of an access transaction T'.
Let T' be an access to the object X. By assumption, serialize(α,T,X) is a schedule of basic object X, and by construction, β|X = serialize(α,T,X). Since $\beta'\pi$|X is a prefix of β|X, $\beta'\pi$|X is a schedule of basic object X. The result follows by Lemma 1.

3) π is CREATE(T')
We must show that π is enabled as an operation of the serial scheduler in state s'. That is, we must show that T' $\in$ s'.create_requested - (s'.created $\cup$ s'.aborted) and that siblings(T') $\cap$ s'.created $\subseteq$ s'.returned.

By the preconditions of the generic controller and Lemma 16, a REQUEST_CREATE(T') event precedes and affects π in α. Since transaction(π)=T', T' must be visible to T in α, and therefore parent(T') is also visible to T in α. Since transaction(REQUEST_CREATE(T'))=parent(T'), the event REQUEST_CREATE(T') occurs in visible(α,T) and thus in β. Since the order of events in β is consistent with affects$_E$(α), the REQUEST_CREATE(T') event must precede π in β, so by Lemma 4, T' $\in$ s'.create_requested. Since only one CREATE for T' occurs in α, no CREATE(T') occurs in β', so by Lemma 4, T' $\notin$ s'.created. Since by Lemma 19, T' is not an orphan in α, no ABORT(T') occurs in α. Thus no ABORT(T') occurs in β', so by Lemma 4 T' $\notin$ s'.aborted. This completes the demonstration that T' $\in$ s'.create_requested - (s'.created $\cup$ s'.aborted).

Now suppose T'' is a sibling of T' that is in s'.created. Then CREATE(T'') occurs in β'. Since the order of events in β is consistent with return$_E$(α), and CREATE(T'') precedes CREATE(T') in β, we cannot have (T',T'') $\in$ return$_T$(α). Thus by Lemma 28, return$_T$(α) contains (T'',T'). Thus α contains a return event ψ for T''. Now transaction(ψ)=parent(T'')=parent(T') is visible to T in α, so ψ occurs in β. Since ψ is an event that mentions T'', we have (ψ,π) $\in$ return$_E$(α), and so ψ precedes π in β. Thus a return event for T'' occurs in β', proving that T'' $\in$ s'.returned.

4) π is COMMIT(T')
We must show that (T',v) $\in$ s'.commit_requested for some v, and that T' $\notin$ s'.returned.

By the preconditions of the generic controller and Lemma 16, there is a value v so that a REQUEST_COMMIT(T',v) precedes and affects π in α. Since π occurs in β, transaction(π)=parent(T') must be visible to T in α. However since COMMIT(T') occurs in α, T' is visible to parent(T') in α, and so by Lemma 9, T' is visible to T in α. Thus REQUEST_COMMIT(T',v) occurs in visible(α,T), and thus it occurs in β. Since the order of events in β is consistent with affects$_E$(α), the REQUEST_COMMIT(T',v) event must occur in β'. Thus (T',v) $\in$ s'.commit_requested.

The other condition, T' $\notin$ s'.returned, is true because (by Lemma 17) there is only one return for π in α and hence only one in β.

5) π is ABORT(T')
We must show that T' $\in$ s'.create_requested - (s'.created $\cup$ s'.aborted) and siblings(T') $\cap$ s'.created $\subseteq$ s'returned.

By the preconditions for the generic controller and Lemma 16, a REQUEST_CREATE(T') event precedes π and affects π in α. Since transaction(REQUEST_CREATE(T')) = parent(T') = transaction(π), the REQUEST_CREATE(T') event occurs in visible(α,T) and hence in β. Since the order of events in β is consistent with affects$_E$(α), the REQUEST_CREATE(T') occurs in β', so that T' $\in$ s'.create_requested. Also T' is an orphan in α, so by Lemma 19, T' is not visible to T in α. Thus CREATE(T') does not occur in visible(α,T), and so also CREATE(T') does not occur in β. Thus T' $\notin$ s'.created. Since by Lemma 17 there is at most one ABORT(T') in α, there can be no ABORT(T') event in β'. Thus by Lemma 4 T' $\notin$ s'.aborted.

Suppose T'' is a sibling of T' such that T'' $\in$ s'.created. By Lemma 4, a CREATE(T'') event occurs in β' preceding π. Since the order of events in β is consistent with return$_E$(α), and CREATE(T'') precedes ABORT(T') in β, we cannot have (T',T'') $\in$ return$_T$(α). Thus by Lemma 28, return$_T$(α) contains (T'',T'). Thus α contains a return event ψ for T''. Now transaction(ψ)=parent(T'')=parent(T') is visible to T in α, so ψ occurs in β. Since ψ is an event that mentions T'', we have (ψ,π) $\in$ return$_E$(α), and so ψ precedes π in β. Thus, T'' $\in$ s'.returned, as required.

6) π is a REPORT_ABORT or REPORT_COMMIT event for T'
By the preconditions of the generic controller and Lemma 16, π is preceded and affected by the appropriate return event ψ in α. Since transaction(ψ)=parent(T')=transaction(π), ψ must occur in visible(α,T) and hence in β. Since the order of events in β is consistent with affects$_E$(α), ψ must occur in β', and so π is enabled as an operation of the serial scheduler after β'. □

The previous lemma enables us to deduce that a generic schedule α is serially correct for a non-orphan transaction T if every object satisfies a simple condition that depends on α and T.

Theorem 32: Let α be a generic schedule, and let T be any transaction that is not an orphan in α. Suppose that for each object X, serialize(α,T,X) is a schedule of basic object X. Then there is a serial schedule β such that $\alpha|T=\beta|T$.

6.2. Dynamic Atomicity

In any practical system, we would hope that all generic schedules would be serially correct for all non-orphan transactions. Let X be a generic object in generic system $\mathcal{S}$. We say that X is *dynamic atomic in* $\mathcal{S}$ if for every schedule α of $\mathcal{S}$, and every transaction T that is not an orphan in α, serialize(α,T,X) is a schedule of basic object X. A generic system $\mathcal{S}$ is a *dynamic atomic system* if every generic object is dynamic atomic in $\mathcal{S}$.

We thus have the following consequence of Theorem 32:

Corollary 33: Dynamic atomic systems are serially correct for every non-orphan transaction.

The Conflict-Based Locking objects we will construct in this paper, and the R/W Locking objects that represent data items implemented using Moss' algorithm, have an even stronger property, because they ensure serial correctness no matter what choices are made of transaction automata to be used in the system. We say a generic object X is *dynamic atomic* if it is dynamic atomic in every generic system (of the fixed system type) that contains X.

Corollary 34: Let S be a generic system such that every generic object is dynamic atomic. Then S is serially correct for every non-orphan transaction.

Thus dynamic atomic objects respond to accesses in such a way that the resulting schedules can be serialized in return order. In fact, the return order is not known to the objects, as it is determined dynamically by the controller, so that a dynamic atomic object must ensure that its responses to accesses can be serialized in every sibling order which could be the return order, based on the local history of the object, but given no information about the other components of the system.

6.3. Local Information

The previous section showed that to prove a system serially correct for non-orphan transactions it is enough to check that each generic object is dynamic atomic. For each generic object X, this is a local condition because it depends only on X, but to check it one must compute serialize(α,T,X) for all schedules α of all generic systems containing X. In this section we introduce a property that is sufficient to prove dynamic atomicity, but that is easily checked merely by examining the schedules of X.

First we introduce some terms that enable us to describe information about the fate of transactions, and about return order, that is available to a generic object given its local history. Let α be a sequence of operations of generic object X, T a transaction, and T' an ancestor of T. We say that T is *committed at X* to T' in α if α contains an INFORM_COMMIT_AT(X)OF(U) for every U that is an ancestor of T and a proper descendant of T'. If α is a sequence of operations of X and T and T' are any transactions, we say that T is *visible at X* to T' in α if T is committed at X in α to lca(T,T'). We denote by visible-at-X(α,T) the subsequence of α consisting of events π such that transaction(π) is visible at X to T in α. We say that T is an *orphan at X* in α if INFORM_ABORT_AT(X)OF(U) occurs in α for some U which is an ancestor of T.

We collect here some obvious facts about visibility at X of transactions.

Lemma 35: Let α be a sequence of operations of X and β a subsequence of α. Let T be an access to X and T' a transaction. If T is visible at X to T' in β then T is visible at X to T' in α.

Lemma 36: Let α be a sequence of operations of X, and let T, T' and T'' be transactions. If

T is visible at X to T' in α, and T' is visible at X to T'' in α, then T is visible at X to T'' in α.

Lemma 37: Let α be a well-formed sequence of operations of X, and let T and T' be transactions. If T is visible at X to T' in α, and T' is not an orphan at X in α, then T is not an orphan at X in α.

We will next define a relation on accesses to the generic object X, which reflects some information that X can deduce about the return order, given its local knowledge of the schedule. Given a sequence α of operations of generic object X, we define a binary relation return-at-$X_T(\alpha)$ on accesses to X, where (U,U') $\in$ return-at-$X_T(\alpha)$ exactly when U $\not\leq$ U', α contains both REQUEST_COMMIT(U,v) and REQUEST_COMMIT(U',v') for some values v and v', and U is visible at X to U' in α', where α' is the longest prefix of α not containing REQUEST_COMMIT(U',v'). The intuition on which this definition is based, is that the COMMIT operation of any ancestor of U must precede an INFORM_COMMIT for that transaction, but the COMMIT for an ancestor of U' must follow the REQUEST_COMMIT for U', so long as U' is not an orphan. Thus if neither U nor U' are orphans, and an INFORM_COMMIT for the ancestor of U that is a child of lca(U,U') precedes the REQUEST_COMMIT for U', then U must be ordered before U' in the return order. The possibility of orphans leads to the complicated definition given.

We will also consider the induced relations return-at-$X_E(\alpha)$ on the events of α, and return-at-$X_D(\alpha)$ on deeds at X. The first is defined by the condition that $(\phi,\pi) \in$ return-at-$X_E(\alpha)$ if and only if ϕ and π are events of α mentioning accesses U and U' respectively, and (U,U') $\in$ return-at-$X_T(\alpha)$. Similarly, we define the second by ((U,v),(U',v')) $\in$ return-at-$X_D(\alpha)$ if and only if $(\phi,\pi) \in$ return-at-$X_E(\alpha)$, where ϕ=REQUEST_COMMIT(U,v) and π=REQUEST_COMMIT(U',v').

We now show that when α is a well-formed sequence of operations of generic object X, return-at-$X_T(\alpha)$ is a partial order on accesses to X. An immediate consequence of this is that the derived relations on events and deeds are also partial orders.

Lemma 38: If α is a well-formed sequence of operations of generic object X then return-at-$X_T(\alpha)$ is a partial order on accesses to X.

Proof: First, we note that by definition return-at-$X_T(\alpha)$ is irreflexive. Now, since α is well-formed, if T $\not\leq$ T' and T is visible at X to T' in α' then α' contains a REQUEST_COMMIT operation for T. Thus the definition shows that (T,T') $\in$ return-at-$X_T(\alpha)$ only if REQUEST_COMMIT(T,v) precedes REQUEST_COMMIT(T',v') in α. Since α is well-formed, it contains at most one REQUEST_COMMIT operation for each access, and therefore return-at-$X_T(\alpha)$ is antisymmetric. Finally, suppose (T,T') $\in$ return-at-$X_T(\alpha)$ and (T',T'') $\in$ return-at-$X_T(\alpha)$. Let α' denote the longest prefix of α not containing REQUEST_COMMIT(T',v') and let α'' denote the longest prefix of α not containing REQUEST_COMMIT(T'',v''). We have seen that REQUEST_COMMIT(T',v') precedes REQUEST_COMMIT(T'',v'') in α, so α' is a prefix of α''. Since T is visible at X to T' in α', T is visible to T' at X in α'', and since T' is visible to T'' at X in α'', we may apply Lemma 36 to deduce that T is visible to T'' at X in α''. Thus (T,T'') $\in$ return-at-$X_T(\alpha)$. □

Lemma 39: If α is a well-formed sequence of operations of generic object X then return-at-$X_E(\alpha)$ is a partial order on the events of α, and return-at-$X_D(\alpha)$ is a partial order on deeds at X.

We say that generic object X is *local-dynamic atomic* if whenever α is a well-formed schedule of X, γ is a sequence of input operations such that $\alpha\gamma$ is well-formed, T is a transaction that is not an orphan at X in $\alpha\gamma$, and β is a reordering of visible-at-X($\alpha\gamma$,T) that is consistent with return-at-$X_E(\alpha)$, and that is a well-formed sequence of operations of basic object X, then β is a schedule of basic object X. We note that return-at-$X_E(\alpha)$ is equal to return-at-$X_E(\alpha\gamma)$. The intuition behind the definition is that α may not contain enough information about the fates of transactions to determine which accesses are visible to T, so we need to consider any sequence γ of input operations that can bring more information. Then the sequences β described will include all sequences that are consistent with the global return order, and so should be considered when proving dynamic atomicity.

We now justify the names introduced above by showing the relationship between each local property defined above and the corresponding global property.

Lemma 40: Let α be a schedule of a generic system containing generic object X. If transaction U is an orphan at X in $\alpha|X$ then U is an orphan in α. Similarly if U is committed at X to V in $\alpha|X$ then U is committed to V in α. Also if U is visible at X to V in $\alpha|X$ then U is visible to V in α.

Proof: These are immediate consequences of the controller preconditions, which imply that any INFORM_ABORT_AT(X)OF(T) in α must be preceded by an ABORT(T), and that any INFORM_COMMIT_AT(X)OF(U) is preceded by COMMIT(U).

Next, we show that for non-orphan transactions, return-at-$X_T(\alpha|X)$ is a subrelation of $\text{return}_T(\alpha)$.

Lemma 41: Let α be a schedule of a generic system containing generic object X. If (T,T') $\in$ return-at-$X_T(\alpha|X)$, and T' is not an orphan in α, then (T,T') $\in$ $\text{return}_T(\alpha)$.

Proof: By definition of the relation return-at-$X_T(\alpha|X)$, $\alpha|X$ contains a REQUEST_COMMIT(T',v') event for some value v'. Letting α' denote the longest prefix of α not containing REQUEST_COMMIT(T',v'), the definition also enables us to say that T is visible at X to T' in α'|X. By Lemma 40, this implies that T is visible to T' in α'. Let U and U' denote the siblings such that T is a descendant of U, and T' is a descendant of U'. Thus α' contains a COMMIT(U) event. However since α is well-formed, it contains at most one REQUEST_COMMIT event for T', and so α' does not contain a REQUEST_COMMIT event for T'. By the controller preconditions, α' does not contain a COMMIT(T') event. Since α is well-formed, α' does contain a CREATE(T') event. Since T' is not an orphan in α, α' does not contain an ABORT(T') event. Thus T' is live in α'. By Lemma 20, U' must be live in α'. Since α' contains a return for U, and no return for U', we have that (U,U') $\in$ return(α). Therefore (T,T') $\in$ $\text{return}_T(\alpha)$. □

Of course it follows immediately that if (π,π') $\in$ return-at-$X_E(\alpha|X)$ and π does not mention a transaction which is an orphan in α, then (π,π') $\in$ return-at-$X_E(\alpha|X)$.

Finally we show that local dynamic atomicity is a sufficient condition for dynamic atomicity.

Lemma 42: If X is a generic object that is local-dynamic atomic then X is dynamic atomic.

Proof: Let S be a generic system containing X as generic object. Let α be a schedule of S, and T a transaction that is not an orphan in α. We must prove that serialize(α,T,X) is a schedule of basic object X. Let γ denote a sequence of operations consisting of an INFORM_COMMIT_AT(X)OF(U) for each COMMIT(U) that occurs in α. Then $\alpha\gamma$ is a schedule of the system S, since each operation in γ is an output of the generic controller which is enabled by Lemma 16. Thus $\alpha\gamma|X$ is well-formed. We note that since INFORM_COMMIT_AT(X)OF(U) occurs in $\alpha\gamma|X$ if and only if COMMIT(U) occurs in α, visible-at-X($\alpha\gamma|X$,T) = visible(α,T)|X. Thus serialize(α,X,T), which is a reordering of visible(α,T) consistent with return$_E$(α), is a reordering of visible-at-X(($\alpha|X$)($\gamma|X$),T) consistent with return-at-X$_E$($\alpha|X$). Furthermore serialize(α,T,X) is a well-formed sequence of operations of basic object X. Because X is local-dynamic atomic, any such reordering must be a schedule of basic object X. This completes the proof that X is dynamic atomic. □

7. Semantic Conditions

We will give an algorithm that performs concurrency control and recovery using information about which locks may not be held simultaneously by different transactions. In order for the algorithm to work correctly, this information about lock "conflicts" must accurately reflect the behavior of the basic object involved. That is, we require operations whose locks do not conflict to have the same effect, regardless of the order in which they occur. To describe this requirement, and to reason about the resulting generic object, we need to introduce some terms to express facts about the semantics of operations. First, we define the fundamental concept of "equieffectiveness" of schedules to capture precisely the idea of two schedules with the same effects. We can then define "commutativity". For our discussion of Read/Write Locking we also need to define "transparency" of operations; an operation is said to be transparent if later accesses to the same object return values that are the same as in the situation where the operation did not occur.

7.1. Equieffective Schedules

We introduce the concept of equieffective schedules of a basic object X in order to define precisely what schedules we will regard as "essentially" the same. Intuitively, these are schedules that leave the automaton in states that are the same. However, we are really interested in observable behavior, not states, so it is enough that they be indistinguishable by later operations. Formally, given two well-formed sequences α and β of operations of X, we say that α is *equieffective*[11] to β if for every sequence ϕ of operations of X such that both $\alpha\phi$ and $\beta\phi$ are well-formed, $\alpha\phi$ is a schedule of X if and only if $\beta\phi$ is a schedule of X. Notice that if neither α nor β is a schedule of X, then α is trivially equieffective to β. Also, notice that if α is equieffective to β and β is a schedule of X, then α is a schedule of X. In the sense of semantic theory, equieffective schedules pass the same tests, where a test involves determining if a given sequence of operations can occur after the sequence being tested. We limit the tests to sequences

[11] This definition was first used in [FLMW]. The definition of well-formedness in that paper was different from the one that was introduced in [LM] and that we use here, and we warn readers that most of the properties proved in [FLMW] about equieffective schedules do not hold for this paper.

that do not violate well-formedness, for technical reasons, because we have not required the objects to behave sensibly if the inputs violate well-formedness. Clearly, α is equieffective to β if and only if β is equieffective to α and in this case we say that α and β are equieffective sequences. We also have an extension result.

Lemma 43: If α and β are equieffective well-formed sequences of operations of X, and ϕ is a sequence of operations of X such that $\alpha\phi$ and $\beta\phi$ are well-formed, then $\alpha\phi$ and $\beta\phi$ are equieffective.

Proof: This is immediate, since well-formed extensions of $\alpha\phi$ are well-formed extensions of α. □

Equieffectiveness is not an equivalence relation, but we do have a restricted transitivity result.

Lemma 44: Let ξ, η and ς be three well-formed sequences of deeds at X, such that every deed in η appears in either ξ or ς. If perform(ξ) is equieffective to perform(η), and perform(η) is equieffective to perform(ς), then perform(ξ) is equieffective to perform(ς).

Proof: Suppose perform(ξ) and perform(η) are equieffective, and that perform(η) and perform(ς) are equieffective. In order to prove that perform(ξ) and perform(ς) are equieffective, we must take any sequence of operations β such that perform(ξ)β and perform(ς)β are well-formed, and show that perform(ξ)β is a schedule of X if and only if perform(ς)β is a schedule. By the definition of equieffectiveness, and the transitivity of "if and only if", it suffices to show that perform(η)β is well-formed. But by Lemma 3, β must be either perform(τ) or perform(τ)CREATE(T), where the first components of all the deeds in τ (and T as well, if appropriate) are distinct from the first components of all the deeds in ξ and ς. By the condition on η, the first components of all the deeds in τ (and T as well, if appropriate) are distinct from the first components of the deeds in η. Thus Lemma 3 completes the proof, by showing that perform(η)β is well-formed. □

7.1.1. Commutativity

Our definition of commutativity is closely based on the one introduced by Weihl for systems without nesting [We]. We extend his work slightly by ignoring the actual state of the data object, and considering only the information about the state that can be determined by later tests.

We say that deeds (T,v) and (T',v') at basic object X *commute* if for any sequence of deeds ξ such that both perform(ξ(T,v)) and perform(ξ(T',v')) are well-formed schedules of X, then perform(ξ(T,v)(T',v')) and perform(ξ(T',v')(T,v)) are equieffective well-formed schedules of X. That is, we can consider for any deed (T,v) the test that determines if a schedule of X can be extended by perform(T,v). Then (T,v) and (T',v') commute if whenever X passes each test, it passes both in any order, and furthermore, no later test can determine which order was used.

To illustrate this definition, we will consider an object X representing a bank account. The accesses to X are of the following kinds:

- balance?: The return value for this access gives the current balance.
- deposit-\$a: This increases the balance by \$a. The only return value is NIL.

- withdraw-\$b: This reduces the balance by \$b if the result will not be negative. In this case the return value is OK. If the result of withdrawing would be to cause an overdraft, then the balance is left unchanged, and the return value is FAIL.

For this object, it is clear that two well-formed schedules that leave the same final balance in the account are equieffective, since the result of each access depends only on the current balance. Now if T and T' are accesses of kind deposit-\$a and deposit-\$b, then the deeds (T,NIL) and (T',NIL) commute. To see this, suppose perform(ξ(T,NIL)) and perform(ξ(T',NIL)) are well-formed schedules of X. This implies that ξ is well-formed and contains no deed with first component T or first component T'. Therefore α = perform(ξ(T,NIL)(T',NIL)) and β = perform(ξ(T',NIL)(T,NIL)) are well-formed. Also since perform(ξ) is a schedule of X, so are each of α and β, since a deposit can always occur. Finally the balance left after each of α and β is \$(x+a+b), where \$x is the balance after perform(ξ), so α and β are equieffective.

On the other hand, let U and U' be distinct accesses of kind withdraw-\$a and withdraw-\$b respectively. Then (U,OK) and (U',FAIL) commute. The reason is that if perform(ξ(U,OK)) and perform(ξ(U',FAIL)) are both well-formed schedules then we must have $a \leq x < b$, where \$x is the balance after perform(ξ). In this situation both perform(ξ(U,OK)(U',FAIL)) and perform(ξ(U',FAIL)(U,OK)) are well-formed schedules of X, which result in a balance of \$(x-a), and so are equieffective. However, (U,OK) and (U',OK) do not commute, since if perform(ξ) leaves a balance of \$x, where $\max(a,b) \leq x < a+b$, then perform(ξ(U,OK)) and perform(ξ(U',OK)) are schedules of X, but perform(ξ(U,OK)(U',OK)) is not a schedule, since after perform(ξ(U,OK)) the balance left is \$(x-a), which is not sufficient to cover the withdrawal of \$b.

This example demonstrates a significant feature of the approach to concurrency control that was taken in [We], and adopted in this paper. We allow the return value of accesses to be considered in determining commutativity, and thus also when deciding whether the accesses can be allowed to proceed concurrently. Traditional database management systems have used an architecture where a lock manager first determines whether an access is to proceed or be delayed, and only later is the response determined. Our approach models both obtaining locks and choosing a response as preconditions on the operation of giving the response. In our example, we show that it is acceptable to allow a successful withdrawal to proceed concurrently with one that fails. By doing so, we obtain concurrency that is unavailable if return values of accesses are not considered in determining commutativity. However we note that our arguments for serial correctness in section 8 merely require that non-commuting accesses be prevented from acting concurrently. They do not require that commuting accesses be allowed to proceed concurrently. Thus our arguments still prove the correctness of traditional systems, where locks are obtained without regard to the value that will be returned.

The next lemma shows how the defining property of commutativity carries through to sequences of deeds.

Lemma 45: If ς and ς' are sequences of deeds at X such that each deed in ς commutes with each deed in ς', and ξ is a sequence of deeds such that both perform($\xi\varsigma$) and perform($\xi\varsigma'$) are well-formed schedules of X, then perform($\xi\varsigma\varsigma'$) and perform($\xi\varsigma'\varsigma$) are equieffective well-formed schedules of X.

7.2. Transparency

We say that a deed (T,v) at X is *transparent* if for any sequence of deeds ξ such that perform(ξ(T,v)) is a well-formed schedule of X, then perform(ξ(T,v)) and perform(ξ) are equieffective well-formed schedules of X.

We extend the defining property to collections of deeds in the following lemma.

Lemma 46: Let η be a sequence of deeds at X such that perform(η) is a well-formed schedule of X, and let ξ be a subsequence of η, such that every deed in η-ξ is transparent. Then perform(η) and perform(ξ) are equieffective well-formed schedules of X.

The next lemma shows how transparency is related to commutativity.

Lemma 47: Let (T,v) and (T',v') be transparent deeds at X such that T $\neq$ T'. Then (T,v) commutes with (T',v').

Proof: Suppose ξ is a sequence of deeds at X such that perform(ξ(T,v)) and perform(ξ(T',v')) are well-formed schedules of X. Therefore no deed in ξ has T or T' as first component, and all the deeds in ξ have distinct first components. Therefore perform(ξ(T,v)(T',v')) and perform(ξ(T',v')(T,v)) are well-formed sequences of operations of X. Now perform(ξ(T,v)) and perform(ξ) are equieffective, since (T,v) is transparent. Since perform(ξ)perform(T',v') is a schedule of X, the definition of equieffectiveness implies that perform(ξ(T,v))perform(T',v')=perform(ξ(T,v)(T',v')) is also a schedule of X. Similarly the fact that (T',v') is transparent implies that perform(ξ(T',v')(T,v)) is a schedule of X. By Lemma 46, each of perform(ξ(T,v)(T',v')) and perform(ξ(T',v')(T,v)) are equieffective to perform(ξ). Lemma 44 now shows that they are equieffective to each other, as required. □

8. Conflict-Based Locking Objects

Given a basic object X and a binary relation CONFLICT on pairs of deeds at X, we construct a generic object automaton W(X) using conflict-based locking, where CONFLICT is used to determine when locks conflict. We show that if CONFLICT contains all pairs of non-commuting deeds, then the conflict-based locking object is dynamic atomic. Note that in many implementations there will be pairs that commute but are nonetheless in CONFLICT (e.g., in exclusive locking, CONFLICT includes all pairs of deeds at X!), but this does not invalidate our correctness proof.

W(X) has the following operations.

Input operations:
CREATE(T), for T an access to X
INFORM_COMMIT_AT(X)OF(T), T $\neq T_0$
INFORM_ABORT_AT(X)OF(T), T $\neq T_0$
Output operations:

REQUEST_COMMIT(T,v), for T an access to X

A state s of W(X) has components s.create_requested, s.run and s.intentions. Of these, create_requested and run are sets of transactions, initially empty, and intentions is a function from transactions to sequences of deeds at X, initially mapping every transaction to the empty sequence Λ. When (T,v) is a member of s.intentions(U), we say that U *holds a (T,v)-lock*. Given a state s and a transaction T we also define the sequence total(s,T) of deeds by the recursive definition total(s,T_0) = s.intentions(T_0), total(s,T) = total(s,parent(T))s.intentions(T). Thus perform(total(s,T)) is the sequence of operations obtained by concatenating the values of intentions along the chain from T_0 to T, and then replacing each (U,u) by CREATE(U)REQUEST_COMMIT(U,u); as we will see, when T is an access to X this records a sequence of operations of basic object X that leads to a state of the basic object which is used as the "current" state when determining the response to T.

The transition relation of W(X) is given by all triples (s',π,s) satisfying the following pre- and postconditions, given separately for each π. As before, any component of s not mentioned in the postconditions is the same in s as in s'.

CREATE(T), T an access to X
Postcondition:
  s.create_requested = s'.create_requested ∪ {T}

INFORM_COMMIT_AT(X)OF(T), $T \neq T_0$
Postcondition:
  s.intentions(T) = Λ
  s.intentions(parent(T)) = s'.intentions(parent(T))s'.intentions(T)
  s.intentions(U) = s'.intentions(U) for $U \neq$ T, parent(T)

INFORM_ABORT_AT(X)OF(T), $T \neq T_0$
Postcondition:
  s.intentions(U) = Λ, U ∈ descendants(T)
  s.intentions(U) = s'.intentions(U), U ∉ descendants(T)

REQUEST_COMMIT(T,v), T an access to X
Precondition:
  T ∈ s'.create_requested - s'.run
  for every U such that U ∉ ancestors(T), and every (T',v') in s'.intentions(U):
    ((T,v),(T',v')) ∉ CONFLICT
  perform(total(s',T)(T,v)) is a schedule of basic object X
Postcondition:
  s.run = s'.run ∪ {T}
  s.intentions(T) = s'.intentions(T)(T,v)
  s.intentions(U) = s'.intentions(U) for $U \neq T$

When an access transaction is created, it is added to the set create_requested. A response containing return value v to an access T can be returned only if the access has been requested but not yet responded to, every holder of a conflicting lock is an ancestor of T, and v is a value such that perform(T,v) can occur as operations of basic object X from a state of basic object X following the schedule perform(total(s',T)). When a response is given, the access transaction is added to run and the deed (T,v) is appended to intentions(T) to indicate that the (T,v)-lock was granted. When a Conflict-Based Locking object is informed of the abort of a transaction, it removes all locks held by descendants of the transaction. When it is informed of a commit, it passes any locks held by the transaction to the parent, appending them at the end of the parents intentions list.

We have the following obvious result, since W(X) has the appropriate operations and preserves well-formedness:

> **Lemma 48:** W(X) is a generic object.

8.1. Properties of W(X)

We first note that the holders of conflicting locks must always be related. This is guaranteed when a lock is granted, and is maintained as locks are passed up from child to parent.

> **Lemma 49:** If α is a well-formed schedule of W(X), s is the state of W(X) after α, T and T' are unrelated transactions, (U,u) is in s.intentions(T), and (U',u') is in s.intentions(T'), then ((U,u),(U',u')) $\notin$ CONFLICT.

The algorithm as described above does not use all the information available to the object about the fate of transactions, because it does not store the fact that an INFORM_COMMIT or INFORM_ABORT operation has occurred. Thus we introduce some terms to describe the information W(X) uses about commits, aborts and return order of transactions after a sequence α of operations of W(X). If α is a sequence of operations of W(X), T is an access to X, and T' is an ancestor of T, we say that T is *lock-committed at X* to T' in α, if α contains a subsequence β consisting of an INFORM_COMMIT_AT(X)OF(U) event for every U that is an ancestor of T and a proper descendant of T', arranged in ascending order (so the INFORM_COMMIT for parent(U) is preceded by that for U). If α is a well-formed sequence of operations of W(X), T is an access to X, and T' is any transaction, we say that T is *lock-visible at X* to T' in α if T is lock-committed at X to lca(T,T'). It is obvious that if T is lock-committed at X to T' in α, then T is committed at X to T' in α. Similarly if T is lock-visible at X to T' in α then T is visible at X to T' in α.

Given a sequence α of operations of W(X), we define a binary relation lock-return-at-$X_T(\alpha)$ on accesses to X, where (U,U') $\in$ return-at-$X_T(\alpha)$ exactly when U $\neq$ U', α contains both REQUEST_COMMIT(U,v) and REQUEST_COMMIT(U',v') for some values v and v', and U is lock-visible at X to U' in α', where α' is the longest prefix of α not containing REQUEST_COMMIT(U',v'). We will also consider the induced relations lock-return-at-$X_E(\alpha)$ on the events of α, and lock-return-at-$X_D(\alpha)$ on deeds at X. The first is defined by the condition that $(\phi,\pi) \in$ lock-return-at-$X_E(\alpha)$ if and only if ϕ and π are events of α

mentioning accesses U and U' respectively, and (U,U') $\in$ lock-return-at-$X_T(\alpha)$. Similarly, we define the second by ((U,v),(U',v')) $\in$ lock-return-at-$X_D(\alpha)$ if and only if (ϕ,π) $\in$ lock-return-at-$X_E(\alpha)$, where ϕ=REQUEST_COMMIT(U,v) and π=REQUEST_COMMIT(U',v'). It is clear that lock-return-at-$X_T(\alpha)$ is a subrelation of return-at-$X_T(\alpha)$.

The following lemma, which can be proved by a straightforward induction, shows which locks are held by a transaction after a schedule of W(X).

Lemma 50: Let α be a well-formed schedule of W(X), and s the state of W(X) reached by applying α to the initial state. Let T be an access to X such that REQUEST_COMMIT(T,v) occurs in α and T is not an orphan at X in α, and let T' be the highest ancestor of T such that T is lock-committed at X to T' in α. Then (T,v) is a member of s.intentions(T'). Conversely, if (U,u) is an element of s.intentions(U') then U is a descendant of U', REQUEST_COMMIT(U,u) occurs in α, and U' is the highest ancestor of U to which U is lock-committed at X in α.

Now for the main result, which shows that, provided the relation CONFLICT includes all non-commuting deeds, certain sequences of operations, extracted from a well-formed schedule of W(X), are well-formed schedules of basic object X. The extra conclusion, that some of these schedules are equieffective, is needed to carry out the induction step of the proof of this lemma. We say that a binary relation CONFLICT on deeds at X is *permissible* if for any two deeds (T,v) and (T',v') at X that do not commute, ((T,v),(T',v')) $\in$ CONFLICT.

Lemma 51: Suppose CONFLICT is a permissible binary relation on deeds at X. Let W(X) be the Conflict-Based Locking object constructed using CONFLICT, and let α be a well-formed schedule of W(X). Let Z be a set of deeds at X such that for all (T,v) $\in$ Z, α contains REQUEST_COMMIT(T,v), T is not an orphan at X in α, and whenever ((T',v'),(T,v)) $\in$ lock-return-at-$X_D(\alpha)$ then (T',v') $\in$ Z. Let ξ and η be total orderings of Z such that each is consistent with lock-return-at-$X_D(\alpha)$. Then perform(ξ) and perform(η) are both well-formed schedules of basic object X. Furthermore, perform(ξ) and perform(η) are equieffective.

Proof: The proof will use induction on the size of the set Z. The basis case, when Z is empty, is trivial. Otherwise, suppose Z contains k deeds, and the lemma has been proved for all sets of k-1 deeds. Let (U,u) denote the last element of ξ, and let Z' = Z - {(U,u)}. Also let $\xi=\xi'$(U,u), and $\eta=\eta_1$(U,u)η_2. Now let s' denote the state of W(X) after α', where α' is the longest prefix of α not containing REQUEST_COMMIT(U,u). Finally, let ς_1=total(s',U), and let ς_2 denote some total ordering, consistent with lock-return-at-$X_D(\alpha)$, of the deeds in Z - (total(s',U) $\cup$ {(U,u)}).

Clearly Z' is a set of k-1 deeds that satisfies the conditions of the lemma, since there is no deed (U',u') in Z such that ((U,u),(U',u')) $\in$ lock-return-at-$X_D(\alpha)$. Also ξ' and $\eta_1\eta_2$ are total orderings of Z' consistent with lock-return-at-$X_D(\alpha)$. Furthermore by Lemma 50 the deeds in total(s',U) are exactly those (U',u') such that ((U',u'),(U,u)) $\in$ lock-return-at-$X_D(\alpha)$, and their order is consistent with lock-return-at-$X_D(\alpha)$. Since for every deed (U',u') in ς_1, and every deed (U'',u'') in ς_2, ((U'',u''),(U',u')) $\notin$ lock-return-at-$X_D(\alpha)$, we have that $\varsigma_1\varsigma_2$ is also a total ordering of Z' consistent with lock-return-at-$X_D(\alpha)$. The induction hypothesis thus shows that perform(ξ'), perform($\eta_1\eta_2$), and perform($\varsigma_1\varsigma_2$) are all equieffective, well-formed schedules of basic object X.

We now begin the proof that perform(ξ) and perform(η) are well-formed schedules of X. We first show that (U,u) commutes with every deed (U",u") in ς_2. There are two possibilities: either REQUEST_COMMIT(U",u") precedes REQUEST_COMMIT(U,u) in α, or else REQUEST_COMMIT(U,u) precedes REQUEST_COMMIT(U",u") in α. In the first case, let V denote the highest ancestor of U" to which U" is lock-committed at X in α'. By Lemma 50, (U",u") $\in$ s'.intentions(V), but by definition of ς_2 V is not an ancestor of U. Therefore, by the preconditions for REQUEST_COMMIT(U,u), which is enabled in state s', we must have that ((U,u),(U",u")) $\notin$ CONFLICT, and therefore in this case (U,u) and (U",u") commute. In the second case, let t denote the state of W(X) after α", the longest prefix of α not containing REQUEST_COMMIT(U",u"). Also let V denote the highest ancestor of U to which U is lock-committed at X in α", so that (U,u) $\in$ t.intentions(V). Now V is not an ancestor of U", as otherwise ((U,u),(U",u")) $\in$ lock-return-at-$X_D(\alpha)$, contradicting the assumption that (U,u) was the last element in ξ. Thus by the preconditions for REQUEST_COMMIT(U",u") as an operation of W(X), ((U",u"),(U,u)) $\notin$ CONFLICT, so in this case also, (U,u) and (U",u") commute.

Note that by the preconditions for REQUEST_COMMIT(U,u) as an operation of W(X). we have that perform(total(s',U)(U,u)) is a schedule of basic object X, which is clearly well-formed, since α is well-formed. That is, perform(ς_1(U,u)) is a well-formed schedule of X. However, we showed above that perform($\varsigma_1\varsigma_2$) is a well-formed schedule of X. Since (U,u) commutes with every deed in ς_2, we have by Lemma 45 that perform($\varsigma_1\varsigma_2$(U,u)) is a well-formed schedule of X. Since we saw that perform($\varsigma_1\varsigma_2$) is equieffective to perform(ξ'), and since perform(ξ)= perform(ξ'(U,u)) is clearly well-formed, Lemma 43 shows that perform(ξ) is a schedule of X that is equieffective to perform($\varsigma_1\varsigma_2$(U,u)). Similarly, since perform(ξ') is equieffective to perform($\eta_1\eta_2$), perform(ξ) is equieffective to perform($\eta_1\eta_2$(U,u)). This completes the proof that perform(ξ) is a well-formed schedule of X. By a symmetrical argument, perform(η) is a well-formed schedule of X.

Since perform(η) is a well-formed schedule of X, we deduce that its prefix perform(η_1(U,u)) is also a well-formed schedule of X. However the induction hypothesis showed that perform($\eta_1\eta_2$) is a well-formed schedule of X. Since every deed in ς_1 is contained in η_1, every deed in η_2 is contained in ς_2, and so (U,u) commutes with every deed in η_2. Therefore perform(η)= perform(η_1(U,u)η_2) is equieffective to perform($\eta_1\eta_2$(U,u)), by Lemma 45. Since perform($\eta_1\eta_2$(U,u)) is equieffective to perform(ξ), we use Lemma 44 to deduce that perform(η) is equieffective to perform(ξ), completing the proof. □

Now we can prove that our algorithm produces local-dynamic atomic generic objects.

Theorem 52: Suppose CONFLICT is a permissible binary relation on deeds at X. Let W(X) be the Conflict-Based Locking object constructed using CONFLICT. Then W(X) is local-dynamic atomic.

Proof: Let $\alpha\gamma$ be a well-formed schedule of W(X), and T a transaction that is not an orphan at X in $\alpha\gamma$. Let β be a reordering of visible-at-X($\alpha\gamma$,T) that is consistent with return-at-$X_E(\alpha)$ and that is a well-formed sequence of operations of basic object X. We must show that β is a schedule of basic object X. Consider the set Z of deeds (U,u) such that REQUEST_COMMIT(U,u) occurs in β, and let ς denote the unique total ordering of Z in which (U,u) precedes (U',u') exactly when REQUEST_COMMIT(U,u) precedes

REQUEST_COMMIT(U',u') in β. Lemma 3 shows that β is equal to either perform(ς) or perform(ς)CREATE(T') for some access T'. We observe that since T' is visible at X to T in $\alpha\gamma$, but $\alpha\gamma$ does not contain a REQUEST_COMMIT operation for T', and so does not contain an INFORM_COMMIT operation for T', we can deduce that T' must be equal to T.

We now show that ς satisfies the conditions given in Lemma 51, and thus that perform(ς) is a schedule of basic object X. That is, suppose (U,u) $\in$ Z. We will demonstrate that $\alpha\gamma$ contains REQUEST_COMMIT(U,u), that U is not an orphan at X in $\alpha\gamma$, and that whenever ((U',u'),(U,u)) $\in$ lock-return-at-$X_D(\alpha\gamma)$, then (U',u') $\in$ Z. We will also demonstrate that the ordering ς is consistent with lock-return-at-$X_D(\alpha\gamma)$, by showing that the latter is a subrelation of the former.

Since (U,u) $\in$ Z, REQUEST_COMMIT(U,u) occurs in β, and thus in visible-at-X($\alpha\gamma$,T) and therefore in $\alpha\gamma$. Since U is visible at X to T in $\alpha\gamma$, and T is not an orphan at X in $\alpha\gamma$, we deduce that U is not an orphan at X in $\alpha\gamma$. If ((U',u'),(U,u)) $\in$ lock-return-at-$X_D(\alpha\gamma)$, then $\alpha\gamma$ must contain REQUEST_COMMIT(U',u') and also U' is lock-visible at X to U in a prefix of $\alpha\gamma$. Thus U' is visible at X to U in $\alpha\gamma$, and therefore U' is visible at X to T in $\alpha\gamma$. We deduce that the REQUEST_COMMIT(U',u') event is in visible-at-X($\alpha\gamma$,T), and therefore in β. That is (U',u') $\in$ Z, as required. We also want to show that if ((U',u'),(U,u)) $\in$ lock-return-at-$X_D(\alpha\gamma)$, then (U',u') precedes (U,u) in ς, that is, that REQUEST_COMMIT(U',u') precedes REQUEST_COMMIT(U,u) in β. Since the order of events in β is consistent with return-at-$X_E(\alpha)$, it suffices to show that $(\phi,\pi) \in$ return-at-$X_E(\alpha)$ where ϕ = REQUEST_COMMIT(U',u') and π = REQUEST_COMMIT(U,u). However if we denote by α' the prefix of α preceding π, then we know that U' is lock-visible at X to U in α', and thus U' is visible at X to U in α', which establishes the fact that ((U',u'),(U,u)) $\in$ return-at-$X_D(\alpha)$, and thus that $(\phi,\pi) \in$ return-at-$X_E(\alpha)$.

Thus the demonstrations in the previous paragraph enable us to deduce that perform(ς) is a schedule of basic object X. Since β is either perform(ς) or perform(ς)CREATE(T), it is now immediate that β is a schedule of basic object X, completing the proof that W(X) is local-dynamic atomic. □

An immediate consequence of Theorem 52, Lemma 42 and Corollary 34 is that if S is a *Conflict-Based Locking system*, that is a generic system in which each generic object is a Conflict-Based Locking object for any permissible choice of CONFLICT relation for that object, then S is serially correct for all non-orphan transactions.

9. R/W Locking objects

We mentioned earlier that the algorithm for general conflict-based locking combines the techniques used by Weihl in the absence of nesting with those used by Moss for Read/Write locking in nested transaction systems. In this section, we recall the R/W Locking object automaton M(X), constructed in [FLMW] as a formal model for an object using Moss' algorithm for concurrency control and recovery in a nested transaction system. We show that if CONFLICT is chosen to include all pairs of deeds except those whose first components are distinct read transactions, then the object W(X) is a suitable specification for M(X), in the sense that any well-formed schedule of M(X) is a well-formed schedule of W(X). In fact, the construction of M(X) is similar to that of W(X), the major difference being that the list of deeds stored in the intentions component of the state of W(X) is replaced by a single version of the state of basic object X, stored in the map component of the state of M(X).

In order to construct M(X) we need a classification of all the accesses to X into two classes, called respectively the *read accesses* and the *write accesses*. We say that this classification is permissible if whenever T is a read access, then (T,v) is a transparent deed at X for any v. If ξ is a sequence of deeds at X, we let write(ξ) denote the subsequence consisting of those deeds whose first components are write accesses. When the classification is permissible, Lemma 46 implies that if perform(ξ) is a well-formed schedule of X, then perform(write(ξ)) is an equieffective well-formed schedule of X.

We first reproduce the construction of M(X) from [FLMW]. M(X) has the following operations.

Input operations:
CREATE(T), for T an access to X
INFORM_COMMIT_AT(X)OF(T), $T \neq T_0$
INFORM_ABORT_AT(X)OF(T), $T \neq T_0$
Output operations:
REQUEST_COMMIT(T,v), for T an access to X

A state s of M(X) consists of the following five components: s.write-lockholders, s.read-lockholders, s.create_requested, and s.run, which are sets of transactions, and s.map, which is a function from s.write-lockholders to states of the basic object X. We say that a transaction in write-lockholders *holds a write-lock*, and similarly that a transaction in read-lockholders *holds a read-lock*. We say two locks *conflict* if they are held by different transactions and at least one is a write-lock. The initial states of M(X) are those in which write-lockholders = $\{T_0\}$ and map(T_0) is an initial state of the basic object X, and the other components are empty. The transition relation of M(X) is given by all triples (s',π,s) satisfying the following pre- and postconditions, given separately for each π. As before, any component of s not mentioned in the postconditions is the same in s as in s'.

```
CREATE(T), T an access to X
Postcondition:
    s.create_requested = s'.create_requested $\cup$ {T}

INFORM_COMMIT_AT(X)OF(T), $T \neq T_0$
Postcondition:
    if T $\in$ s'.write-lockholders then
        begin
            s.write-lockholders = (s'.write-lockholders - {T}) $\cup$ {parent(T)}
            s.map(U) = s'.map(U) for U $\in$ s.write-lockholders - {parent(T)}
            s.map(parent(T)) = s'.map(T)
        end
    if T $\in$ s'.read-lockholders then
        begin
        s.read-lockholders = (s'.read-lockholders - $\{T\}$) $\cup$ $\{$parent(T)$\}$
    end

INFORM_ABORT_AT(X)OF(T), $T \neq T_0$
Postcondition:
    s.write-lockholders = s'.write-lockholders - descendants(T)
    s.read-lockholders = s'.read-lockholders - descendants(T)
    s.map(U) = s'.map(U) for all U $\in$ s.write-lockholders

REQUEST_COMMIT(T,v) for T a write access to X
Precondition:
    T $\in$ s'.create_requested - s'.run
    s'.write-lockholders $\cup$ s'.read-lockholders $\subseteq$ ancestors(T)
    there are states t, t' of basic object X so that
        (s'.map(least(s'.write-lockholders)),CREATE(T),t)
        and (t,REQUEST_COMMIT(T,v),t')
        are in the transition relation of basic object X
Postcondition:
    s.run = s'.run $\cup$ $\{T\}$
    s.write-lockholders = s'.write-lockholders $\cup$ $\{T\}$
    s.map(U) = s'.map(U) for all U $\in$ s.write-lockholders - $\{T\}$
    s.map(T) = t'

REQUEST_COMMIT(T,v) for T a read access to X
Precondition:
    T $\in$ s'.create_requested - s'.run
    s'.write-lockholders $\subseteq$ ancestors(T)
    there are states t, t' of basic object X so that
        (s'.map(least(s'.write-lockholders)),CREATE(T),t)
        and (t,REQUEST_COMMIT(T,v),t')
        are in the transition relation of basic object X
Postcondition:
    s.run = s'.run $\cup$ $\{T\}$
    s.read-lockholders = s'.read-lockholders $\cup$ $\{T\}$
```

It is clear that a R/W Locking object preserves well-formedness, and so is a generic object.

When an access transaction is created, it is added to the set create-requested. A response containing return value v to an access T can be returned only if the access has been requested but not yet responded to, every holder of a conflicting lock is an ancestor of T, and v is a value that can be returned by basic object X in the response to T from a state obtained by performing CREATE(T) in the state that is the value of map at the least member of write-lockholders. When a response is given, the access transaction is added to run and granted the appropriate lock, and if the transaction is a write access, the resulting state is stored as map(T). If the transaction is a read access, no change is made to the stored state of the basic object X, i.e. to map.

When a R/W Locking object is informed of the abort of a transaction, it removes all locks held by descendants of the transaction. When it is informed of a commit, it passes any locks held by the transaction to the parent, and also passes the version stored in map, if there is one.[12]

We use the same terms (with the same definitions) to describe which transactions are *lock-committed at X*, *aborted at X*, etc. that we used for W(X).

Here are some simple facts about the state of M(X) after a schedule α, each easily proved by induction on the length of the schedule.

> **Lemma 53:** Let α be a schedule of M(X), and s a state of M(X) reached by applying α to an initial state. Suppose T $\in$ s.write-lockholders and T' $\in$ s.read-lockholders $\cup$ s.write-lockholders. Then either T is an ancestor of T' or else T' is an ancestor of T.
>
> **Lemma 54:** Let α be a well-formed schedule of M(X), and s a state of M(X) reached by applying α to an initial state. If T is an access to X such that REQUEST_COMMIT(T,v) occurs in α and T is not an orphan at X in α, let T' be the highest ancestor of T such that T is lock-committed at X to T' in α. Then if T is a write access, T' must be a member of s.write-lockholders, while if T is a read access, T' must be a member of s.read-lockholders. Conversely, if T' is a member of s.write-lockholders, then there is some write access T to X such that REQUEST_COMMIT(T,v) occurs in α, T is not an orphan at X in α, and T' is the highest ancestor of T such that T is lock-committed at X to T' in α. Similarly if T' is a member of s.read-lockholders, then there is some read access T to X such that REQUEST_COMMIT(T,v) occurs in α, T is not an orphan at X in α, and T' is the highest ancestor of T such that T is lock-committed at X to T' in α.

9.1. The relationship between M(X) and W(X)

Now we construct a relation CONFLICT between deeds at X, by defining ((T,v),(T',v')) $\in$ CONFLICT unless T and T' are distinct read accesses to X. We consider the conflict-based locking object W(X) constructed using this relation. If the classification of accesses used by M(X) is permissible, then Lemma 47 implies that any pair of deeds that do not commute are in the relation, hence CONFLICT is permissible and so W(X) is dynamic atomic. The following results show how the behavior of M(X) is related to that of W(X). The first lemma is straightforward, since the postconditions for each operation of M(X) are similar to the postconditions for the same operation of W(X). The second establishes the correspondence between the state of X stored by M(X) in map, and the schedule of X stored by W(X) in intentions. The third is the result we want, which shows that in an environment that ensures well-formedness (in particular in a generic system) any behavior of M(X) is a possible behavior for W(X) provided M(X) is constructed using a permissible classification. Thus M(X) is local-dynamic atomic if the classification used is permissible.

[12] If the reader wishes to compare our version of the algorithm with that in [Mo], the following may be useful: Moss gives the name "the associated state" for object X and transaction T to what we call s.map(T') where T' is the least ancestor of T in s.write-lockholders, and he calls s.map(least(s.write-lockholders)) "the current state" of X. Also, he removes a read-lock when the owner also holds a write-lock (this is an optimization that does not affect the correctness proof). Moss also allows internal transactions to directly access objects, whereas we allow only leaf transactions to perform data access.

Lemma 55: Let α be a well-formed schedule of M(X) that is also a well-formed schedule of W(X). Let s be a state of M(X) reached by applying α to an initial state of M(X), and let t be the unique state of W(X) reached by applying α to the initial state of W(X). Then s.create_requested=t.create_requested, s.run=t.run, s.write-lockholders is the set of T such that t.intentions(T) contains a deed whose first component is a write access to X, and s.read-lockholders is the set of T such that t.intentions(T) contains a deed whose first component is a read access to X.

Lemma 56: Let M(X) be constructed using a permissible classification of accesses to X. Let α be a well-formed schedule of M(X) that is also a well-formed schedule of W(X). Let s be a state of M(X) reached by applying α to an initial state of M(X), and let t be the unique state of W(X) reached by applying α to the initial state of W(X). Then for every transaction T perform(write(total(t,T))) is a well-formed schedule of basic object X that can leave X in the state s.map(T'), where T' is the least ancestor of T such that T' $\in$ s.write-lockholders.

Proof: We use induction on the length of α. The basis case is trivial, so let $\alpha = \alpha'\pi$, where α' is a well-formed schedule of M(X). Let s' denote a state of M(X) after applying α' such that (s',π,s) is a step of M(X). Also let t' denote the state of W(X) after α'. There are five cases, for each of which we will relate s.map to s'.map, and perform(write(total(t,T))) to perform(write(total(t',T))).

(1) π is CREATE(U) for an access U to X.
The postconditions of π as an operation of M(X) and W(X) show that s.map=s'.map, s.write-lockholders=s'.write-lockholders, and t.intentions=t'.intentions. Thus T' is also the least ancestor of T in s'.write-lockholders, and so the induction hypothesis says that perform(write(total(t',T))) is a well-formed schedule of X that can leave X in state s'.map(T'). That is, perform(write(total(t,T))) is a well-formed schedule of X that can leave X in state s.map(T').

(2) π is REQUEST_COMMIT(U,u) for U a read access to X.
Examining the postconditions for π as an operation of M(X) and W(X) we see that s.map(W)=s'.map(W) for all W, and s.write-lockholders = s'.write-lockholders, and also t.intentions(W)=t'.intentions(W) unless W=U. Thus T' is the least ancestor of T in s'.write-lockholders, and s.map(T')=s.map(T'). Also, if T $\neq$ U, total(t,T)=total(t',T). On the other hand, if U=T, total(t,T)=total(t',T)(U,u), so write(total(t,T))=write(total(t',T)). In either case, perform(write(total(t,T)))=perform(write(total(t',T)))), which is, by the induction hypothesis, a well-formed schedule of X that can leave X in state s'.map(T')=s.map(T'), as required.

(3) π is REQUEST_COMMIT(U,u) for U a write access to X.
Examining the postconditions for π as an operation of M(X) and W(X) we see that s.map(W)=s'.map(W) unless W=U, and s.write-lockholders = s'.write-lockholders $\cup$ {W}, and also t.intentions(W)=t'.intentions(W) unless W=U. Thus if U $\neq$ T, T' is the least ancestor of T in s'.write-lockholders, and s.map(T')=s'.map(T'). Also, if T $\neq$ U, total(t,T)=total(t',T), and so the induction hypothesis shows that perform(write(total(t,T)))=perform(write(total(t',T)))) is a well-formed schedule of basic object X that can leave X in state s'.map(T')=s.map(T'). On the other hand, if U=T, total(t,T)=total(t',U)(U,u), so perform(write(total(t,T))) = perform(write(total(t',U)))perform(U,u). Since π is enabled as an operation of M(X) in state s', perform(U,u) can take place starting from state s'.map(U') where U'=least(s'.write-lockholders), and the postconditions for π ensure that s.map(U) is a state that can result from this application. Since all members of s'.write-lockholders must be ancestors of U, U' is the least ancestor of U in s'.write-lockholders, and so the induction hypothesis implies that

perform(write(total(t',U))) is a well-formed schedule of basic object X that can leave X in state s'.map(U').. Combining these facts, we see that when U=T, perform(write(total(t,T))) is a well-formed schedule of X that can leave X in state s.map(U)=s.map(T), as required.

(4) π is INFORM_COMMIT_AT(X)OF(U)
Now t.intentions(W)=t'.intentions(W) unless W=U or W=parent(U). Similarly s.map(W)=s'.map(W) unless W=U or W=parent(U). The discussion is divided into subcases, depending on the relation of T and U in the transaction tree.

(i) U is an ancestor of T.
If U is the least ancestor of T in s'.write-lockholders then by the definition of M(X), T' must be parent(U) and s.map(T') = s'.map(U), while if U is not the least ancestor of T in s'.write-lockholders then T' must be the least ancestor of T in s'.write-lockholders and s.map(T') = s'.map(T'). In either case, s.map(T') is s'.map(T''), where T'' is the least ancestor of T in s'.write-lockholders. Also total(t,T)=total(t',T). The desired result follows immediately from the inductive hypothesis.

(ii) U is not an ancestor of T, but parent(U) is an ancestor of T.
Here we give separate arguments, depending on whether U is in s'.write-lockholders or not. If U ∈ s'.write-lockholders then Lemma 53 implies that no ancestor of T that is a strict descendant of parent(U) can be in s'.write-lockholders or in s'.read-lockholders. The definition of M(X) therefore shows that T' = parent(U) and that s.map(T') = s'.map(U). Also we note that some deed in t'.intentions(U) must be a deed of a write access, and that t'.intentions(W) must be empty for all W that are ancestors of T and strict descendants of parent(U). Therefore total(t,T)=total(t,parent(U))=total(t',U). By the induction hypothesis perform(write(total(t',U))) is a well-formed schedule of X that can leave X in state s'.map(U), that is perform(write(total(t,T))) is a well-formed schedule of X that can leave X in state s.map(T).

On the other hand, if U ∉ s'.write-lockholders then s.write-lockholders = s'.write-lockholders and s.map = s'.map. Thus T' is the least ancestor of T in s'.write-lockholders, and s.map(T')=s'.map(T'). Also no deed in t'.intentions(U) can be a deed of a write access. Since total(t,T) is formed from total(t',T) by the insertion of t'.intentions(U) after the prefix total(t',parent(U)), write(total(t,T))=write(total(t',T)). Thus perform(write(total(t,T))) = perform(write(total(t',T))) is, by the induction hypothesis, a well-formed schedule of basic object X that can leave X in state s'.map(T')=s.map(T') as required.

(iii) parent(U) is not an ancestor of T.
Then T' is the least ancestor of T in s'.write-lockholders and s.map(T') = s'.map(T'). Also total(t,T)=total(t',T). The desired result follows immediately from the inductive hypothesis.

(5) π is INFORM_ABORT_AT(X)OF(U).
We distinguish two subcases, according to the relationship between U and T. If U is an ancestor of T, then total(t,T)=total(t,parent(U))=total(t',parent(U)), as t.intentions(U') is empty for all U' descended from U, but is equal to t'.intentions(U') otherwise. By the induction hypothesis, perform(write(total(t,T))) = perform(write(total(t',parent(U)))) is a well-formed schedule of X that can leave X in state s'.map(T''), where T'' is the least ancestor of parent(U) in s'.write-lockholders. However, since s.write-lockholders=s'.write-lockholders - descendants(U), T'' is also the least ancestor of T in s.write-lockholders, thus T''=T'. Since s.map(T'')=s'.map(T''), as T'' is not a descendant of U, this completes the proof of the lemma in this subcase.

If U is not an ancestor of T, then total(t,T)=total(t',T), so by the induction hypothesis perform(write(total(t,T))) is a well-formed schedule of X that can leave X in state s'.map(T')=s.map(T"), since T' is also the least ancestor of T in s'.write-lockholders. □

Lemma 57: Let M(X) be constructed using a permissible classification of the accesses to X and let α be a well-formed schedule of M(X). Then α is a well-formed schedule of W(X).

Proof: Since the definition of well-formedness is the same for all generic objects, we need only show that α is a schedule of W(X). We use induction on the length of α. The basis case is trivial, so let $\alpha = \alpha'\pi$, where α' is a well-formed schedule of M(X). If π is an input to W(X), the Input Condition implies that α is a schedule of W(X). Thus we have only two other cases to consider.

(1) π is REQUEST_COMMIT(U,u) for U a read access to X.
Let s' denote a state of M(X) after applying α' such that π is enabled as an operation of M(X) after s'. Let t' denote the state of W(X) after α'. We wish to show that π is enabled as an operation of W(X) in state t'. That is, we need to show that U ∈ t'.create_requested - t'.run, that whenever U' ∉ ancestors(U) and (V,v) is a deed in t'.intentions(U') then ((U,u),(V,v)) ∉ CONFLICT, and that perform(total(t',U)(U,u)) is a schedule of basic object X. Since π is enabled as an operation of M(X) in state s', we have that U ∈ s'.create_requested - s'.run, that whenever U' ∈ s'.write-lockholders then U' ∈ ancestors(U), and that perform(U,u) can be performed by X from state s'.map(least(s'.write-lockholders)).

Lemma 55 shows that t'.create_requested = s'.create_requested, and t'.run = s'.run. Therefore we deduce that U ∈ t'.create_requested - t'.run.

Lemma 55 also shows that when V is a write access such that (V,v) is a deed in t'.intentions(U'), then U' ∈ s'.write-lockholders, and hence U' must be an ancestor of U. Also, since U is a read access, the construction of the CONFLICT relation shows that ((U,u)(V,v)) ∈ CONFLICT implies that either U=V, or else V is a write access. Therefore, if (V,v) is a deed in t'.intentions(U') (so U' is an ancestor of V) and ((U,u),(V,v)) ∈ CONFLICT, the facts just proved show that U' must be an ancestor of U. Equivalently, if U' is not an ancestor of U, and (V,v) is a deed in t'.intentions(U') then ((U,u),(V,v)) ∉ CONFLICT.

Finally, let U'= least(s'.write-lockholders). We note that U' must be an ancestor of U and is thus the least ancestor of U in s'.write-lockholders. Therefore Lemma 56 implies that perform(write(total(t',U))) is a schedule of X that can leave X in state s'.map(U'). Then we can deduce that perform(write(total(t',U)))perform(U,u) is a schedule of X. Since perform(write(total(t'.U))) is equieffective to perform(total(t',U)), perform(total(t',U))perform(U,u) = perform(total(t',U)(U,u)) is a schedule of basic object X, since it is well-formed. This completes the proof that α is a schedule of W(X).

(2) π is REQUEST_COMMIT(U,u) for U a write access to X.
Let s' denote a state of M(X) after applying α' such that π is enabled as an operation of M(X) after s'. Let t' denote the state of W(X) after α'. We wish to show that π is enabled as an operation of W(X) in state t'. The proof that U ∈ t'.create_requested - t'.run, and that perform(total(t',U)(U,u)) is a schedule of basic object X, is identical to that in case (1) above. We also must show that whenever U' ∉ ancestors(U) and (V,v) is a deed in t'.intentions(U') then ((U,u),(V,v)) ∉ CONFLICT. We will prove the equivalent statement that if t'.intentions(U') is not the empty sequence, then U' ∈ ancestors(U). By Lemma 55, if (V,v) ∈ t'.intentions(U') and V is a write access then U ∈ s'.write-lockholders, while if (V,v) ∈ t'.intentions(U') and V is a read access then U ∈ s'.read-lockholders. However, since π is enabled as an operation of M(X) in state s', we have that whenever U' ∈ s'.read-lockholders ∪

s'.write-lockholders then U' ∈ ancestors(U), completing the demonstration that α is a schedule of W(X). □

Corollary 58: Let M(X) be constructed using a permissible classification of accesses at X. Then M(X) is local-dynamic atomic.

An immediate consequence of Corollary 58, Lemma 42, and Corollary 34 is that if S is a *R/W Locking system*, that is a generic system in which each generic object is a R/W Locking object for any permissible classification of accesses to that object, then S is serially correct for all non-orphan transactions. This was the main result of [FLMW]. Furthermore, we note that it is always permissible to classify all accesses as write accesses. If that is done, Moss' algorithm degenerates into Exclusive Locking. Thus our results also imply the correctness of Exclusive Locking systems, which was the main theorem of [LM] (albeit with slight differences in the definitions of the system components).

10. Conclusions and Further Work

We have used I/O automata to provide clear formal descriptions of all the components of a nested transaction system. We have demonstrated that any schedule of a Conflict-Based Locking system is serially correct for every non-orphan transaction. We have also shown that our new conflict-based algorithm can be combined with Moss' algorithm using read- and write-locks. Indeed, we have shown that a nested transaction system is serially correct so long as each data object has a simple local property called dynamic atomicity. We have shown how to take any schedule of such a system, extract a subsequence (including all the operations of a given non-orphan transaction), and rearrange the events in the subsequence to give a serial schedule.

This work by no means exhausts the topic of concurrency control and recovery in nested transaction systems. Recent work on timestamp-based concurrency control [As] uses arguments very similar to those in this paper. We hope in the near future to combine these results in a uniform framework. We also hope to extend the results of this paper in several ways. One natural extension is to locking protocols for abstract data types that are built by combining atomic entities of primitive type, for example a queue built from atomic elements. Such protocols have been studied in transaction systems without nesting [We]. We expect that it should be possible to prove that the combined type is dynamic atomic if the individual entries are.

Other aspects of real systems that we have not addressed in this paper, but expect to study in the future, are liveness properties and resilience to system crashes. We have proved that any response to a Conflict-Based Locking system is correct, but for a practical system we also need to know that a response will be produced (and, we hope, rapidly produced.) Lynch and Tuttle [LT] discuss how to express liveness results in terms of I/O automata, but phenomena such as deadlock in transaction systems make it difficult to guarantee strong liveness properties. At best, any guarantees that progress can be made will be probabilistic. System crashes, that cause information (such as lock tables) to be lost, are also a reality of practical systems. We plan to extend the model presented in this paper to describe crashes, and to analyze algorithms that ensure resilience to crashes.

11. Acknowledgements

We thank the members of the Theory of Distributed Systems seminar at MIT for many helpful suggestions.

12. References

[A] Allchin, J.E., "An Architecture for Reliable Decentralized Systems", Ph.D. Thesis, School of Info. and Comp. Sci., Georgia Institute of Technology, September 1983.

[As] Aspnes, J., "Timestamp Ordering and Nested Transactions," M.S. Thesis, MIT Laboratory for Computer Science, Cambridge, MA., June 1987.

[BBG] Beeri, C., Bernstein, P. A., and Goodman, N., "A Model for Concurrency in Nested Transaction Systems," Technical Report, Wang Institute TR-86-03, March 1986.

[BBGLS] Beeri, C., Bernstein, P. A., Goodman, N., Lai, M. Y., and Shasha, D. E., "A Concurrency Control Theory for Nested Transactions," Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing, 1983, pp. 45-62.

[BG] Bernstein, P. A., and Goodman, N., "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys* 13,2 (June 1981), pp. 185-221.

[D] Davies, C. T., "Recovery Semantics for a DB/DC System," *Proc. ACM National Conference 28*, 1973, pp. 136-141.

[EGLT] Eswaren, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L., "The Notions of Consistency and Predicate Locks in Database Systems," *Communications of the ACM*, Vol. 19, No. 11, November 1976, pp. 624-633.

[FLMW] Fekete, A., Lynch, N., Merritt, M., and Weihl, W., "Nested Transactions and Read/Write Locking," Proceedings of 6th ACM Symposium on Principles of Database Systems, 1987, pp. 97-111. An expanded version is available as Technical Memo MIT/LCS/TM-324, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA., April 1987.

[GL] Goldman, K., and Lynch, N., "Nested Transactions and Quorum Consensus," Proceedings of 6th ACM Symposium on Principles of Distributed Computation, 1987, pp. 27-41. An expanded version is available as Technical Report MIT/LCS/TR-390, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA., May 1987.

[Go] Goree, J., "Internal Consistency Of A Distributed Transaction System With Orphan Detection," MS Thesis, TR-286, Laboratory for Computer Science, MIT, January 1983.

[Gr] Gray, J., "Notes on Database Operating Systems," in Bayer, R., Graham, R. and Seegmuller, G. (eds), Operating Systems: an Advanced Course, Lecture Notes in Computer Science, Vol. 60, Springer-Verlag, 1978.

[HLMW] Herlihy, M., Lynch, N., Merritt, M., and Weihl, W., "On the Correctness of Orphan Elimination Algorithms," Proceedings of 17th IEEE Symposium on Fault-Tolerant Computing, 1987, pp. 8-13.

[Ho] Hoare, C. A. R., "Communicating Sequential Processes," Prentice Hall International, 1985.

[Ko] Korth, H. F., "Deadlock Freedom Using Edge Locks," *ACM Trans. on Database Systems*, Vol. 7, No. 4, December 1982, pp. 632-652.

[KS] Kedem, Z., and Silberschatz, A., "Non-two phase locking protocols with shared and exclusive locks," *Proc. Int. Conference on Very Large Data Bases*, 1980, pp. 309-320.

[LHJLSW] Liskov, B., Herlihy, M., Johnson, P., Leavens, G., Scheifler, R., and Weihl, W., "Preliminary Argus Reference Manual," Programming Methodology Group Memo 39, October 1983.

[LiS] Liskov, B., and Scheifler, R., "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", *ACM Transactions on Programming Languages and Systems* Vol. 5, No. 3, July 1983, pp. 381-404.

[LM] Lynch, N., and Merritt, M., "Introduction to the Theory of Nested Transactions," Technical Report MIT/LCS/TR-367, MIT Laboratory for Computer Science, Cambridge, MA., July 1986.

[LT] Lynch, N., and Tuttle, M., "Hierarchical Correctness Proofs for Distributed Algorithms," Proceedings of 6th ACM Symposium on Principles of Distributed Computation, 1987, pp. 137-151. An expanded version is available as Technical Report MIT/LCS/TR-387, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA., April 1987.

[Ly] Lynch, N. A., "Concurrency Control For Resilient Nested Transactions," *Advances in Computing Research* 3, 1986, pp. 335-373.

[MGG] Moss, J. E. B., Griffeth, N. D., and Graham, M. H., "Abstraction in Concurrency Control and Recovery Management" Technical Report 86-20, COINS University of Massachussetts, Amherst, MA., May 1986.

[Mo] Moss, J. E. B., "Nested Transactions: An Approach To Reliable Distributed Computing," Ph.D. Thesis, Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, Cambridge, MA., April 1981. Also, published by MIT Press, March 1985.

[P] Papadimitriou, C. H., "The Serializability of Concurrent Database Updates," *J.ACM* Vol. 26, No. 4, October 1979, pp.631-653.

[R] Reed, D. P., "Naming and Synchronization in a Decentralized Computer System," Ph.D Thesis, Technical Report MIT/LCS/TR-205, MIT Laboratory for Computer Science, Cambridge, MA 1978.

[T] Thomas, R. H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Trans. on Database Systems*, Vol. 4, No. 2, June 1979, pp. 180-209.

[We] Weihl, W. E., "Specification and Implementation of Atomic Data Types," Ph.D Thesis, Technical Report/MIT/LCS/TR-314, MIT Laboratory for Computer Science, Cambridge, MA., March 1984.

OFFICIAL DISTRIBUTION LIST

Director — 2 Copies
Information Processing Techniques Office
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

Office of Naval Research — 2 Copies
800 North Quincy Street
Arlington, VA 22217
Attn: Dr. R. Grafton, Code 433

Director, Code 2627 — 6 Copies
Naval Research Laboratory
Washington, DC 20375

Defense Technical Information Center — 12 Copies
Cameron Station
Alexandria, VA 22314

National Science Foundation — 2 Copies
Office of Computing Activities
1800 G. Street, N.W.
Washington, DC 20550
Attn: Program Director

Dr. E.B. Royce, Code 38 — 1 Copy
Head, Research Department
Naval Weapons Center
China Lake, CA 93555

Dr. G. Hooper, USNR — 1 Copy
NAVDAC-OOH
Department of the Navy
Washington, DC 20374

END
FEb.
1988
DTIC